

Pascal-2[®]

Deel I

Version 2.1 for RT-11

User Manual

Second Edition
August 1983

**Oregon
Software**

I will

Oregon Software

The software described by this publication is subject to change without notice. Oregon Software assumes no responsibility for the use or reliability of any of its software that is modified without the prior written consent of Oregon Software.

Oregon Software holds right, title, and interest in the software described herein. The software, or any copies thereof, may not be made available to or distributed to any person or installation without the written approval of Oregon Software.

This publication, or parts of it, may be copied for use with the licensed software described herein, provided that all copies include this notice and all copyright notices.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-division (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013 of the Federal Acquisitions Regulations (FARs).

Name of Contractor and Address:

Oregon Software, Inc.
6915 S.W. Macadam Avenue
Portland, Oregon 97219
Phone: 503-245-2202

© 1985, 1984, 1983 Oregon Software, Inc. All Rights Reserved. Printed in USA, January 1985.

Pascal-1, Pascal-2 and Oregon Software are trademarks of Oregon Software, Inc.

DEC, PDP, RSX, RSTS and RT-11 are trademarks of Digital Equipment Corporation.

TeX is a trademark of the American Mathematical Society.

TSX-Plus is a trade mark of S&H Computer Systems, Inc.

UPDATE No. 2

Pascal-2 V2.1 User Manual

Second Edition

January 1985

Synopsis: This update package contains pages to be inserted in the *Pascal-2 User Manual* for RT. These new pages reflect the status of the software in Version 2.1D. This update notice should be kept in the front of your manual as a history of changes to the manual.

©1985 by Oregon Software, Inc.

INSTRUCTIONS

New and updated pages are indicated by the date at the bottom of the page. The nature of the changes, i.e., their substance, is indicated by the summary in the table below.

Remove the outdated pages of the user manual and insert the enclosed pages as listed below.

<u>Remove</u>	<u>Insert</u>	<u>Purpose</u>
Title page, ii iii to viii xi 1-1, 1-2	Title page, ii iii to viii xi 1-1, 1-2	Replaced copyright message. Replaced the old table of contents with a new one. Changed name and address in Preface. Replaced reference to Jensen and Wirth's <i>Pascal User Manual and Report</i> with Doug Cooper's <i>Standard Pascal User Reference Manual</i> .
2-1, 2-2	2-1, 2-2	Added sentence explaining that your system manager may change the name of the PASCAL invocation command.
2-11 to 2-14.4	2-11 to 2-14.4	Replaced old "External Modules" with new "External Modules."
2-19, 2-20	2-19, 2-20	Corrected errors in the "Support Library" section.
2-29, 2-30	2-29, 2-30	Removed reference to <code>\$\$HEAP</code> in <code>NewOK</code> Listing.
2-43, 2-44	2-43, 2-44	Made minor editing changes.
2-49 to 2-56	2-49 to 2-56	Moved section "Calling FORTRAN Subroutines" to "External Modules."
3-9, 3-10	3-9, 3-10	Corrected lexical representation of <i>structured-constant</i> . Described prohibited use of <i>structured-constant</i> as <i>case</i> label.
3-17 to 3-20	3-17 to 3-20	Altered explanation of "Extended-Range Arithmetic." Corrected minor spelling and reference errors.
3-31, 3-32	3-31, 3-32	Corrected reference to <i>forward</i> as non-standard Pascal.
3-39, 3-40	3-39, 3-40	Corrected lexical representation of <i>structured-constant</i> .
4-1 to 4-30	4-1 to 4-36	Replaced old "Debugger Guide" with new "Debugger Guide."
5-17 to 5-20	5-17 to 5-20	Changed description of <i>max-len</i> . Changed some procedure and function names.
5-49, 5-50	5-49, 5-50	Corrected default value of <i>L</i> .
Index(all)	Index(all)	Added new entries.

MEMORANDUM

TO : THE SECRETARY OF THE ARMY

FROM : THE CHIEF OF STAFF

SUBJECT: [Illegible]

1. [Illegible text]

2. [Illegible]

3. [Illegible]

4. [Illegible text]

5. [Illegible text]

6. [Illegible text]

7. [Illegible text]

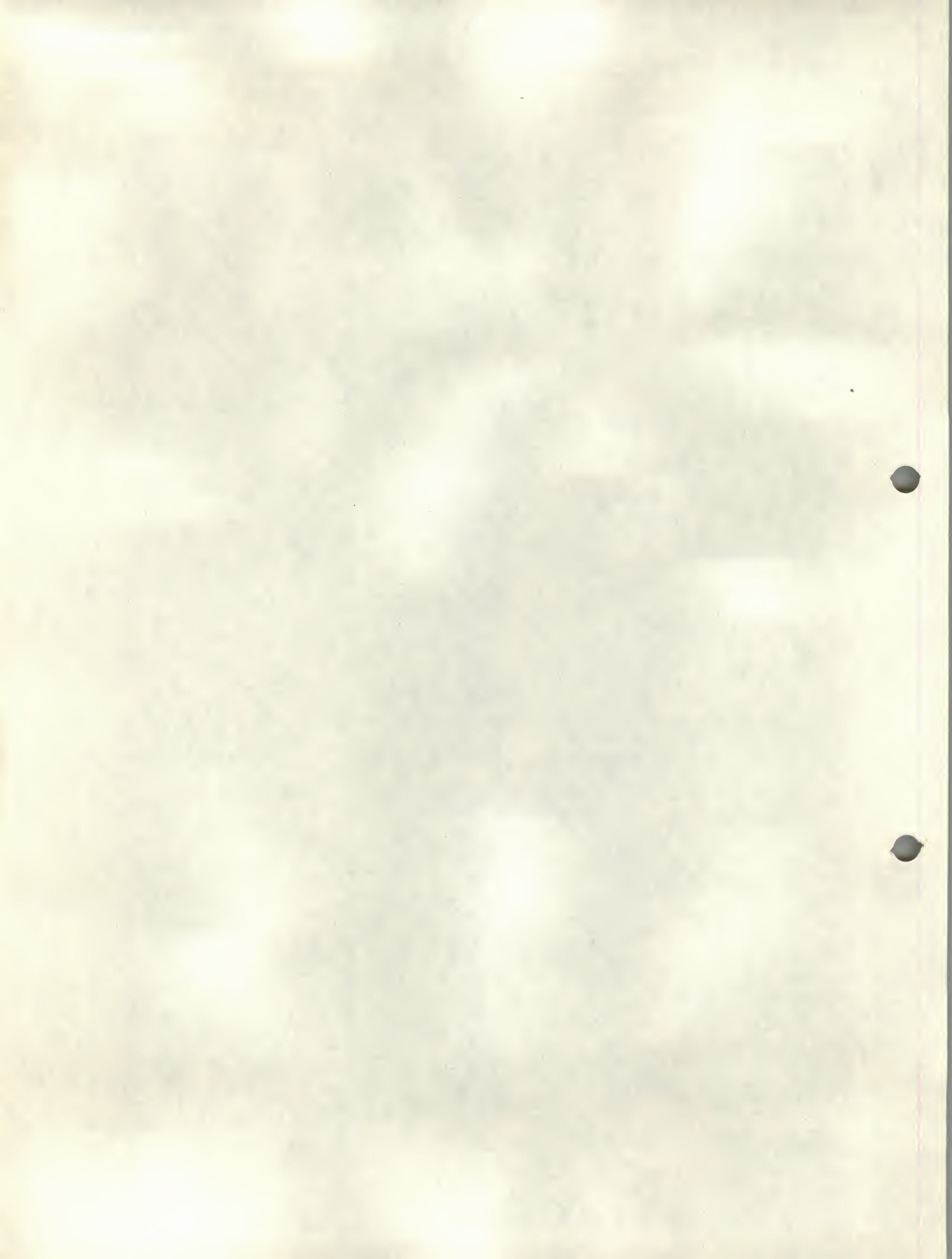
8. [Illegible text]

9. [Illegible text]

10. [Illegible text]

11. [Illegible text]

12. [Illegible text]



Contents

	<u>Page</u>
Preface	xi
Thanks to...	xi
Pascal-2 V2.1/RT-11 Introduction	xiii
About Version 2.1	xiv
Pascal-2 Documentation Package	xv
Style Notes	xvi
Support Policy	xvii
Pascal-2 V2.1/RT-11 User Guide	1-1
Introduction to the User Guide	1-1
Getting Started	1-1
Compiling the Program	1-1
Checking For Errors	1-2
Errors Detected at Run-Time	1-3
Compilation Options	1-4
The Program Listing	1-4
The Formatter	1-4
The Debugger	1-6
Double Precision	1-8
The Profiler	1-8
Your Next Step	1-9
Pascal-2 V2.1/RT-11 Programmer Reference	2-1
Introduction	2-1
Compiler Commands	2-1
Compilation Switches	2-2
Program Options	2-2
Compiler Options	2-2
Code Switches	2-3
Checking Switches	2-3
Processor Switches	2-3
Embedded Switches	2-4
Program Options	2-5
Compiler Options	2-5
Run-Time Checking Switches	2-7
Compilation Examples	2-7
Linking and Executing	2-8
I/O Control Switches	2-9
External Modules	2-11
Calls to Pascal-2 Routines	2-12
Example Using External Directive	2-12
Calls to Non-Pascal-2 Routines	2-14
Calling MACRO Subroutines	2-14
Calling FORTRAN Subroutines	2-14.2
External Module Libraries	2-14.4
The Linker, Overlays, and the Librarian	2-15
The Linker	2-15
Virtual Jobs and the XM Monitor	2-15
Overlays	2-16
Low Memory Overlays	2-16

Extended Memory Overlays	2-17
Linking a Program Whose Root Exceeds 16K	2-17
The Librarian	2-18
Extended Precision	2-18
Support Library	2-19
Initializing the Support Library	2-19
Support Library Data Definitions	2-19
Support Library's Use of Channels	2-21
Run-Time Organization	2-22
Form of the Generated Code	2-22
Memory Organization	2-23
RT-11 System Area	2-23
Program Code	2-23
Global Variables	2-23
Constants	2-23
Tables	2-23
Run-Time Library	2-23
The Stack	2-24
The Heap	2-24
The Monitor and I/O Page	2-25
The Stack Frame	2-25
Function Return Value	2-26
Parameters	2-26
Return Link	2-26
Dynamic Link	2-26
Local Variables	2-26
Register Save Area	2-26
Temporary Storage	2-26
Monitoring Memory Usage	2-27
The 'Space' Function	2-28
Function 'P\$inew' and Procedure 'P\$dispose'	2-28
Example: Function 'NewOK'	2-30
Storage Allocation	2-33
Run-Time Error Reporting	2-35
Examples	2-36
I/O Error Trapping	2-37
Procedure 'SayErr'	2-38.1
Customizing Error Reporting	2-39
Error Termination Status	2-43
Implementation Notes	2-44
Multiple Source Files	2-44
Local Files Closed on Procedure Exit	2-45
Specifying the Location of the Compiler's Work Files	2-46
Variable Initialization	2-46
Reading Command Lines	2-46
Foreground Operation	2-49
Lazy I/O	2-50
Incorporating Lazy I/O into V2.0 Programs	2-51
Terminal I/O	2-51
Single-Character or 'ODT' Mode	2-52
Function 'ReadSn'	2-53
Random Access to 'Text' Files	2-54
Procedure 'GetPos'	2-54
Procedure 'SetPos'	2-55

Unsigned Integer Conversion	2-57
Compiler Optimizations	2-59
Variable Assignments to Registers	2-59
Assignment of Constants and Addresses to Registers	2-59
Constant Folding	2-59
Dead Code Elimination	2-59
Boolean Expression Optimization	2-59
Expression Targeting	2-60
Common Subexpression Elimination	2-60
Common Branch Tail Elimination	2-60
Array Index Simplification	2-60
Appendix A: Compilation Error Messages	2-61
Appendix B: Run-Time Error Messages	2-72
Appendix C: Compiler Errors	2-76
Overflow Errors	2-76
Consistency Checks	2-76
Appendix D: Default File Extensions	2-77
Appendix E: Entry Points in the Pascal Support Library	2-78
Pascal-2 V2.1/RT-11 Language Specification	3-1
Introduction to the Language Specification	3-1
Changes in the Standard	3-1
'For' Statement Control Variables	3-1
File Declaration	3-1
Parameter Compatibility	3-1
Procedure and Function Parameters	3-2
Conformant Array Parameters	3-2
Literal Strings	3-4
'Write,' 'Writeln' of 'Packed Array of Char'	3-4
Identifiers	3-5
Alternate Symbol Representations	3-5
Implementation Definitions	3-6
Standard Type 'Integer'	3-6
Standard Type 'Real'	3-6
Standard Type 'Char'	3-6
Standard Type 'Text'	3-6
'Set' Types	3-6
I/O Definitions	3-7
Syntax Extensions	3-8
Identifiers	3-8
Program Heading	3-8
Declaration Order	3-8
'%Include' Lexical Directive	3-8
'%Page' Lexical Directive	3-9
'External' and 'NonPascal' Directives	3-9
Structured Constants	3-9
Examples of Structured Constants	3-10
Default Case Label ('Otherwise')	3-13
I/O Support Extensions	3-14
External File Access	3-14
'Close' Procedure	3-15
Random Access to Data Files ('Seek')	3-15
String Input ('Read' and 'Readln')	3-16
'Break' Procedure	3-16
Octal Output	3-16

Real Number Formatting	3-17
Low-Level Interface	3-18
Boolean Operators on Integer	3-18
Nondecimal Integer Constants	3-18
Extended-Range Arithmetic	3-18
"Origin" Declaration	3-20
'Ref' Function	3-21
'Size' and 'Bitsize' Functions	3-21
'Loophole' Function	3-21
Non-Standard Language Elements	3-25
Program Parameters	3-25
Directives	3-25
'Mod' of Negative Numbers	3-25
'Eof' Not Accurate for Binary Files	3-25
Returning of Structured Types	3-25
Additional Predefined Functions and Procedures	3-26
Procedure 'Delete'	3-26
Procedure 'Rename'	3-26
Predefined Function 'Time'	3-27
Procedure 'TimeStamp'	3-28
Error Handling	3-30
Detected Errors	3-30
Undetected Errors	3-31
Appendix A: Predefined Identifiers	3-32
Appendix B: Reserved Words	3-32
Appendix C: Pascal-2 Syntax	3-33
Pascal-2 Syntax Diagrams	3-33
Extended Backus-Naur Form	3-38
Pascal-2 Lexical Description	3-39
Pascal-2 EBNF Syntax	3-40
Pascal-2 V2.1/RT-11 Debugger Guide	4-1
Including the Pascal-2 Debugger in Your Program	4-2
Identifying Pascal Statements	4-2
Controlling the Debugger	4-4
Command Syntax	4-4
Exiting and Stopping the Debugger	4-5
Selective Debugging	4-5
Breakpoint Commands	4-6
B, B(): Control Breakpoints	4-6
K, K(): Killing of Breakpoints	4-6
V, V(): Data Breakpoints (Variables)	4-7
Execution Control Commands	4-8
G: Go	4-8
C, C(): Continue Execution	4-8
Examples of the B, K, D, G and C Commands	4-8
S, S(): Step to Next Statement	4-9
P, P(): Proceed to Next Statement	4-9
Examples of the S and P Commands	4-10
Tracking Commands	4-11
H, H(): History of Program Execution	4-11
T(): Execution Trace	4-11
Example of the T Command	4-11
Data Commands	4-13
W(): Write Variable Value	4-13

Variable Assignment	4-14
Examples of the W Command and Variable Assignment	4-15
Informational Commands	4-17
D: Display Parameters	4-17
L, L(): List Source Lines	4-17
Utility Commands	4-18
M(): Define Macro	4-18
X(): Execute Macro	4-18
Examples of the M and X Commands	4-19
Execution Stack Commands	4-20
H, H(): History of Program Execution	4-20
N, N(): Names of Variables	4-21
E(): Enter Stack-Frame Context	4-21
Examples of the H, N, and E Commands	4-22
Stepping Through a Debugger Session	4-23
Debugging External Modules	4-26
Differences in the Commands	4-26
Example	4-27
Overlays	4-30
Appendix A: Debugger Command Summary	4-31
The Pascal-2 Profiler	4-32
Pascal-2 V2.1/RT-11 Utilities Guide	5-1
Introduction to the Utilities Guide	5-1
PASMAT: A Pascal-2 Formatter	5-2
Overview of Capabilities	5-2
Comments	5-2
Statement Bunching	5-3
Tables	5-3
Using PASMAT	5-3
Formatting Directives	5-4
Limitations and Errors	5-6
PASMAT Examples	5-7
PB: A Pascal-2 Formatter	5-9
Using PB	5-9
Example	5-10
Detailed Formatting Rules	5-12
XREF: A Pascal-2 Cross-Reference Lister	5-13
Using XREF	5-13
Limitations	5-13
Example of XREF Listing	5-14
PROCREF: Pascal-2 Procedural Cross-Reference Lister	5-15
Using PROCREF	5-15
Limitations	5-16
Example	5-16
Dynamic String Package	5-18
The Procedures and Functions	5-19
Example	5-20
MACRO-11 Procedures With Pascal-2	5-23
Design of MACRO-11 Procedures	5-23
The PASMAT Macro Package	5-24

Using PASMAL	5-25
Procedure Definition Macros	5-26
The 'Proc' Macro	5-26
The 'Func' Macro	5-27
The 'Param' Macro	5-27
The 'Var' Macro	5-28
The 'Save' Macro	5-28
The 'Rsave' Macro	5-29
The 'Begin' Macro	5-29
The 'Endpr' Macro	5-30
Type Definitions	5-31
Example	5-33
Placing PASMAL into the System Macro Library	5-36
PROSE: A Text Formatter	5-38
PROSE Basics	5-39
Structure of Directive Lines	5-40
Placement of Directives	5-40
Running the PROSE Program	5-42
Header Files	5-42
Controlling Input to PROSE	5-43
INPUT Directive	5-43
OPTION Directive	5-44
Setting Up the Document's Format	5-47
Page Format	5-47
FORM Directive	5-47
Page Breaks	5-48
Margins	5-49
Paragraphs	5-49
Comments	5-51
Changing Format Within the Text	5-51
Breaking and Skipping Lines	5-51
Keep Buffers	5-52
Reset Directive	5-53
Creating an Index	5-54
Printing the Document	5-55
Specifying Output Devices	5-55
Printing Selected Pages and Sections	5-56
Appendix A: Examples of PROSE Directives in Text	5-57
Appendix B: Summary Directive Table	5-63
Appendix C: Historical Notes	5-64
For More Information	Info-1
Index	Index-1

List of Figures

<u>Figure</u>	<u>Page</u>
-- The Pascal-2 Software Development System	xiii
2-1 Typical Memory Layout of a Pascal Program	2-24
2-2 Format of a Stack Frame	2-25
2-3 Tracking Memory Usage with the SPACE Function	2-29

Preface

This edition of the *Pascal-2 User Manual* for the RT-11 operating system corresponds to the latest software release and describes all of the features of Version 2.1 of the Pascal-2 software. This manual is printed looseleaf to allow us to send future errata as Update Packages with "change pages" to be inserted in the existing manuals. Update Packages are sent with new releases when changes made to the software require new or different explanations in the manual. New releases of software, including a set of release notes, are sent to licensed users of the Pascal-2 compiler approximately every six months.

We are grateful to the many readers whose thoughtful and perceptive suggestions have helped us improve our manuals. Please fill out and return the evaluation report provided at the end of this manual, or address your comments directly to:

David Spencer, Manager
Technical Publications
Oregon Software
6915 SW Macadam Ave.
Portland, Oregon 97219

We appreciate hearing from you.

Thanks to...

Oregon Software got its start at OMSI—the Oregon Museum of Science and Industry. OMSI is a private educational organization chartered "to enhance the general public understanding of science and technology, with a strong commitment to education," particularly of youth. It was in the Research Laboratory at OMSI that we began writing software. Seven of us came from OMSI to found Oregon Software in September, 1977. Because of the close association, the name "OMSI" stayed with us for a while, and we continue to support OMSI and its educational programs.

As part of our own research, we've been using and distributing T_EX, a text-processing system developed by Don Knuth at Stanford University. This publication is typeset in the Computer Modern Roman family of type faces with the T_EX system. Draft versions were produced at Oregon Software on an Imprint-10 laser printer driven by T_EX-in-Pascal and our VAX-11/780.

Special thanks to Barbara Beeton and Monty Nichols for their encouragement and assistance with T_EX and to David Kellerman and Barry Smith for implementing T_EX at Oregon Software.

1. The first part of the paper discusses the importance of the study and the objectives of the research. It also mentions the scope of the study and the limitations of the study.

2. The second part of the paper discusses the methodology used in the study. It includes the data collection methods and the data analysis methods.

3. The third part of the paper discusses the results of the study. It includes the findings of the study and the conclusions drawn from the study.

4. The fourth part of the paper discusses the implications of the study. It includes the practical implications of the study and the theoretical implications of the study.

5. The fifth part of the paper discusses the limitations of the study. It includes the limitations of the study and the suggestions for future research.

6. The sixth part of the paper discusses the conclusion of the study. It includes the main findings of the study and the overall conclusion of the study.

7. The seventh part of the paper discusses the references. It includes the list of references used in the study.

8. The eighth part of the paper discusses the appendix. It includes the list of appendixes used in the study.

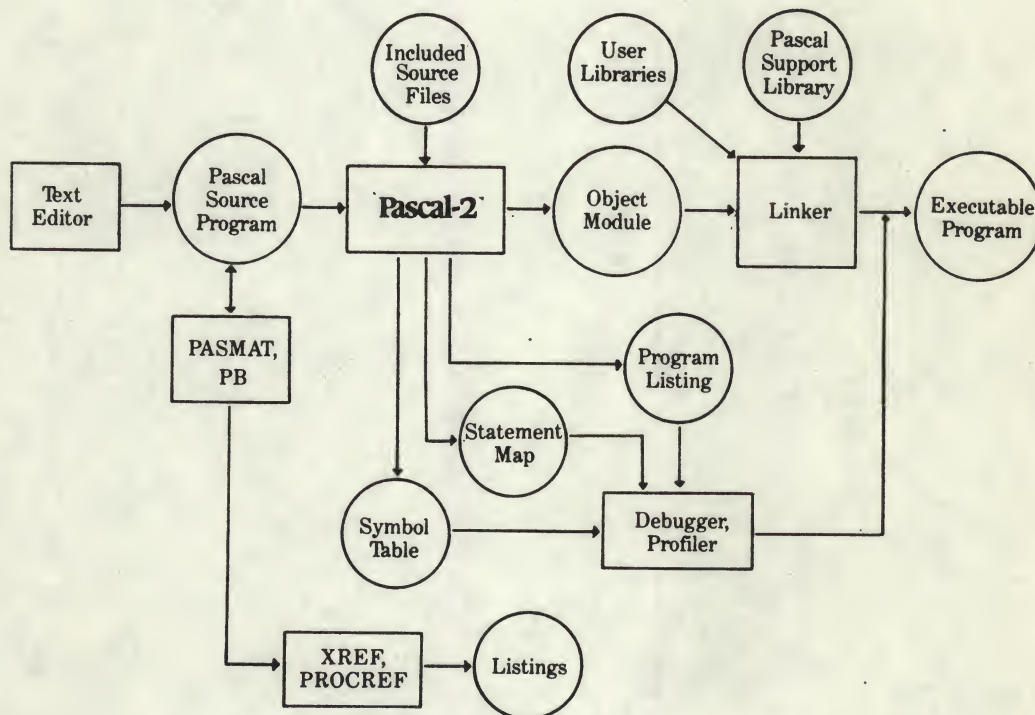
9. The ninth part of the paper discusses the bibliography. It includes the list of bibliographies used in the study.

10. The tenth part of the paper discusses the index. It includes the list of index entries used in the study.

Pascal-2 V2.1/RT-11 Introduction

Pascal-2 is an integrated system for software development. At the heart of the system is a transportable multipass compiler that adheres to the Pascal standard while performing optimizations to generate compact, fast code. The Pascal-2 system also offers sophisticated error checking during compilations, extensive error reporting and recovery at run-time, a Debugger to examine the dynamic state of a running program in a high-level Pascal context, plus other development utilities. Together, these components offer the professional programmer a structured and unified environment in which to design, code, test, maintain, and improve software. The result should be the production of more reliable programs in less time than with other programming packages. Further, use of the Pascal-2 compiler will allow programs to be transported to other computer systems with a minimum of change, thereby accelerating future software development.

Developed over several years, Pascal-2 grew out of our experience with our first Pascal compiler, Pascal-1. Pascal-1 is a one-pass compiler specific to Digital Equipment Corporation's PDP-11 series, with low-level extensions giving the programmer control over the PDP-11 hardware and operating system. Pascal-2 is larger and compiles more slowly than Pascal-1, but Pascal-2 produces code that is much smaller and faster than Pascal-1 code. Typical programs are 30 to 40 percent smaller and up to twice as fast.



The Pascal-2 Software Development System

The Pascal-2 system consists of the Pascal-2 compiler, the support library, the formatters PASMAT and PB, the Debugger and Profiler, and the cross-references XREF and PROCREF. The text formatter PROSE, not shown, is also a component of the system. The user creates the Pascal source program, the included source files, and user libraries. The Text Editor and Linker are supplied by the computer vendor.

Handwritten text, likely bleed-through from the reverse side of the page. The text is illegible due to fading and the quality of the scan.

Handwritten text, likely bleed-through from the reverse side of the page. The text is illegible due to fading and the quality of the scan.

Handwritten text, likely bleed-through from the reverse side of the page. The text is illegible due to fading and the quality of the scan.

Pascal-2 V2.1/RT-11 Introduction

About Version 2.1

Version 2.1 of the Pascal-2 software represents the first major technical improvement in the Pascal-2 software since its release in 1981. The product now includes conformant array parameters, raising the compiler to Level 1 of the proposed international standard. The implementation of lazy I/O changes the way Pascal-2 performs input operations, where input is not read until it is actually needed. Numerous compiler and support library bugs have been fixed. Run-time error diagnostics now include a procedure-by-procedure walkback from the point of error to the main program and are capable of trapping I/O errors, allowing users to recover from I/O errors with their own code. Users may change the wording of run-time error messages and print additional information about the error for debugging purposes.

Enhancements of the Pascal-2 software are summarized in the following list; detailed explanations of each feature appear in the appropriate sections of the manual.

Summary of Changes, New Features in V2.1 Software

<u>Feature</u>	<u>Function</u>
Lazy I/O	An I/O scheme in which data is not read until actually used in the program.
Conformant Array Parameters	Allows you to write general procedures that accept array parameters of different size and with different lower and upper bounds.
Run-Time Errors	Additions include new error numbers and messages, and explanations of those messages. User processing of run-time errors gives you control of run-time error reporting. You may change the wording of run-time error messages and print additional information about the error. The run-time error walkback aids in diagnosis of run-time errors, pinpointing the location of the error in Pascal source terms and showing the reverse sequence of procedure calls leading to the error.
'Nowalkback' Switch	Disables the error walkback. Available as either compilation or embedded switch. This switch is "off" by default (the walkback is printed).
I/O Error Trapping	You can now recover from I/O errors with your own code.
'Workspace:n' Switch	Reduces/increases the work space required by the compiler to compile a program.
Support Library	Library entry points now have the form P\$nnn , where <i>nnn</i> is an integer in the range 0..135. Most routines are now called in the same way as other Pascal procedures, using the standard Pascal calling sequence.
Predefined Procedures	New procedures getpos and setpos simulate random access to files of type text . Function space determines the amount of stack and heap available to an executing program. New procedures rename and delete allow you to rename or delete files from within a program.
Files	Files opened local to a procedure are now closed upon procedure exit. Also, the WK: option allows you to specify the device to which the compiler directs its working data files.
Eight-Bit Characters	Allows you to use extended character sets.

Summary of Changes, New Features in V2.1 Software (cont.)

<u>Feature</u>	<u>Function</u>
Non-Decimal Integer Constants	Allows the use of integers in radices ranging from base 2 (binary) to base 16 (hexadecimal).
Size Restrictions	Easing of certain restrictions allows larger programs to be compiled.
Expanded Unsigned Integers and Functions	The unsigned integer range is 0..65535. Unsigned functions are capable of returning unsigned and structured values.
String Package	Includes the new routines <code>equal</code> , <code>assign</code> and <code>assignchar</code> . Implemented with conformant array parameters. Procedure <code>delete</code> is now named <code>deletestring</code> , to prevent conflict with the new predefined procedure <code>delete</code> .
'%Include' Directive	New syntax allows specification of the disk volume number of the included file.

Because the code generated by the V2.1 compiler differs from that of V2.0, current Pascal-2 users must recompile existing Pascal-2 programs and external modules with the new compiler. Some programs may need to be changed in order to compile or redesigned to take advantage of features such as conformant array parameters, lazy I/O and user control of run-time error reporting.

Pascal-2 Documentation Package

The Pascal-2 user documentation contains information on the use of the Pascal-2 compiler and related utilities on Digital's RT-11 operating systems: RT-11 V4 and V5, SJ, XM, and TSX-Plus. In general, we assume that readers of the manual are programmers familiar with Pascal and the RT-11 operating system. Some sections assume a detailed working knowledge of the language.

The *Pascal-2 User Manual* is not intended to be a Pascal textbook. Beginners can make their way carefully through this manual, but we refer you to the reading list in the appendix, "For More Information."

The manual consists of five major guides, as follows:

- The User's Guide serves as a quick overview of the Pascal-2 system, to give you a feel for how it works. The guide, written on a beginner's level, takes you through the basic steps of compiling, correcting, and running a Pascal-2 program. The User's Guide also has brief explanations and examples of some of the standard features and utilities of the Pascal-2 system.
- The Programmer's Guide contains detailed descriptions of compilation commands, embedded and low-level switches, and the low-level interface between Pascal-2 and the operating system. The Programmer's Guide also contains a miscellany of information on implementation-related problems, divided into two broad categories: error recovery and implementation notes. Finally, the guide describes Pascal-2's optimizations and provides helpful hints as to the cause of compile-time and run-time errors and ways to fix the errors.

10. 10. 1944

Dear Mr. [Name],
I have received your letter of the 10th inst. and am
glad to hear that you are interested in the
[Project Name]. I am sure that the [Project Name]
will be of great value to you and I am sure
that you will find it very interesting.

I am sure that you will find it very interesting
and I am sure that you will find it very
interesting. I am sure that you will find it
very interesting and I am sure that you will
find it very interesting. I am sure that you
will find it very interesting and I am sure
that you will find it very interesting.

I am sure that you will find it very interesting
and I am sure that you will find it very
interesting. I am sure that you will find it
very interesting and I am sure that you will
find it very interesting. I am sure that you
will find it very interesting and I am sure
that you will find it very interesting.

- The Language Specification describes Pascal-2's language features in detail. Since the second edition of Jensen and Wirth's *User Manual and Report* in 1978, the language has undergone major changes, which are incorporated in the Third ISO Draft Proposal 7185. Because not everyone is familiar with that document, the Language Specification begins by summarizing those changes and describing the ways that Pascal-2 implements them. Thus, the guide serves not only as a description of our Pascal product but also as a review of the language's evolution since 1978.
- The Debugger and Profiler Guide describes two programs designed to alleviate tedious aspects of programming or to improve the usefulness of the Pascal-2 system. The Debugger helps find and correct errors that cannot be caught at compile time. The execution Profiler shows areas of the program in which the most time is spent.
- The Utilities Guide describes each of the following packages: program formatters, a text formatter, cross-reference programs, a package that helps interface assembler routines with Pascal-2 programs, and a dynamic string package. Each utility is described in detail, with examples.

Included with the manual is a set of release notes including the Installation Guide.

For information on the RT-11 system, see these RT-11 manuals: *Introduction to RT-11*, *System User's Guide*, *Software Support Manual*, *System Utilities Guide*.

In addition, Pascal-1 customers upgrading to Pascal-2 should refer to the *Pascal-2 Conversion Guide* and the CONVRS utility, which are available from Oregon Software. The *Conversion Guide* explains specific language differences between Pascal-1 and Pascal-2 and the practical programming problems created by the differences. The guide describes the use of the CONVRS utility to help isolate areas in a Pascal-1 program that will have to be modified to convert to Pascal-2; the guide then details the steps required to convert the programs. The *Conversion Guide* concludes with a list of solutions to errors that you may encounter while completing the conversion to Pascal-2.

Style Notes

The *Pascal-2 User Manual* follows these style conventions:

Text:

Pascal reserved words, predefined symbols, switches and compiler directives are in boldface typewriter type: **begin**, **write**, **%include**, **nomain**. Portions of examples referred to in text are in boldface typewriter type. System directives are in upper-case boldface typewriter type: **.SETTOP**, **.CALFIP**. Program and system names are in upper case: **ROTAT**, **RT-11**.

Program Examples:

Commands that you should type are in underlined boldface typewriter: **RUN EX**. These commands assume a carriage return at the end.

Program Listings:

The Pascal-2 compiler accepts any combination of upper-case and lower-case characters. Examples in this manual have Pascal words in lower case and have user-defined words with an initial capital letter and other capitalization as needed for readability, as shown in this program fragment:

```
procedure Show;
begin
    SomeUserAction;
    writeln(Result);
end;
```

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the above matter.

I am sorry to hear that you are not satisfied with the result of the investigation.

I have been very busy lately, and have not had time to devote to this matter as much as I would like.

I am sure that you will understand my position, and that I am doing the best I can for you.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

Enclosed find a copy of the report of the committee on the subject of the above matter.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

Single quotes ('.') in examples and in text appear as '..'.

Terminology:

We use standard terms as they are used in documents describing the RT-11 operating system.

Support Policy

The license fee for your Pascal-2 system includes one year of software support, which covers the following:

- Telephone assistance. We'll provide a quick cure to your problem if at all possible.
- Formal, written response to all problems, suggestions, and comments received in writing. For complex problems, we need written descriptions of your technical problem to ensure correct diagnosis and repair. (This service does **not** include applications consultation.)
- A no-cost update to the latest revision of the Pascal-2 system, upon the written request of your Designated Contact Person. This is the standard response to bugs that have been fixed. (We do charge for the media used for the update.)
- The Oregon Software Pascal Newsletter, which contains status reports on all of our Pascal products, announcements of new versions of software and new products, and various technical articles.

Support may be renewed annually. Customers of an Oregon Software distributor should contact their distributor for support.

Handwritten text, likely a letter or document, written in cursive script. The text is faint and mostly illegible due to fading. It appears to be a formal communication, possibly a letter of introduction or a business document. The text is organized into several paragraphs, with some lines indented. The handwriting is elegant but difficult to decipher.

Pascal-2 V2.1/RT-11 User Guide

Introduction to the User Guide

This is the introductory section, the User Guide. It explains:

- How to compile and run Pascal programs;
- How to interpret program listings and error messages;
- Some details of the compilation process.

This guide assumes that you are familiar with:

- Simple RT-11 commands;
- A text editor (e.g., EDIT, TECO, KED);
- Elementary Pascal programming.

This guide is not:

- An introduction to Pascal (see *Programming in Pascal* by Peter Grogono);
- A detailed description of Pascal-2 (see the Language Specification, and Doug Cooper's *Standard Pascal User Reference Manual*);
- An expert's guide to Pascal-2 (see the Programmer's Guide).

Getting Started

The first step in running a Pascal program is to enter the program into the computer and store it in the file system. Use a familiar text editor to enter the program; store it in a file with the extension .PAS. The Pascal-2 compiler accepts free-format program files, so use blanks, tabs, new lines, and form feeds as desired to help make the program readable.

This Pascal version of a program is called the source program, or the source file. All other versions of the program are translations from the source program.

Compiling the Program

After editing, you must compile the program—translate it into a form that the computer can execute—and link it to the Pascal-2 support library. With the compiler and the support library on the system disk and with a source file called TEST.PAS, the entire compilation process follows this example:

```
.R PASCAL  
•TEST
```

```
.LINK TEST.SY:PASCAL
```

As the example shows, the .PAS extension may be omitted from file names on commands to the Pascal-2 system but must be included in commands to other RT-11 systems such as the editor.

Handwritten header or title at the top of the page.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Pascal-2 V2.1/RT-11 User Guide

To illustrate the compilation process, let's say that the program

```
program First (output);
begin
  write ('Things are best in their beginnings');
  writeln (' -- Blaise Pascal');
end.
```

is stored in the file FIRST.PAS.

Compilation proceeds as follows:

```
.R PASCAL
*FIRST
```

```
LINK FIRST.SY:PASCAL
```

```
.RUN FIRST
```

```
'Things are best in their beginnings' -- Blaise Pascal
```

Notice that no errors were detected. The next example shows what happens if detectable errors are present in the source program.

Checking For Errors

The Pascal-2 compiler detects nearly 150 types of "grammatical" errors in a program: errors in syntax such as missing semicolons, undefined identifiers, missing `begin` and `end` reserved words, and similar mistakes. As an example, the following program contains a deliberate error: a semicolon is missing between the program heading and the reserved word `begin`.

```
program Second (output)
begin
  writeln ('Things get worse as they continue');
end.
```

Semicolon errors (the most common errors made by beginning Pascal programmers) are always detected by the compiler:

```
.R PASCAL
*SECOND
```

```
Pascal-2 RT11 SJ V2.1D  9-Feb-84  7:08 AM  Site #1-1  Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
SECOND
```

```
1      program Second (output)
```

```
~19
```

```
*** 19: Use ';' to separate statements
```

```
*** There was 1 line with errors detected ***
```

```
?Errors detected: 1
```

For each detected error, a line of the source program is printed, then an arrow indicating the approximate position of the error, then a message describing the error. (The number "19" is the error message number generated by the compiler.) See Appendix A of the Programmer's Guide for a complete list of detectable compilation errors.

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

Errors Detected at Run-Time

The errors discussed so far have been compilation errors — errors detected by the compiler. Run-time errors, on the other hand, occur when a program is executing, after it has been compiled and linked.

A run-time error such as “array subscript out of bounds” stops the program at the point of error. The Pascal-2 error reporting system prints header information and the error message, then traces the program’s execution history, procedure by procedure, from the point of error back to the main program. The error traceback, or “walkback,” is intended to make debugging easier by showing precisely where the program stopped and which procedures were called to reach that point.

The following is an example of a run-time error and procedure walkback. (The program has already been compiled and linked.) Line numbers appearing in the walkback correspond to line numbers in the source listing, not line numbers in individual procedures.

```
.R PASCAL
*CUSTOM
```

```
PASCAL--Fatal error at user PC= 7244B
Array subscript out of bounds
```

```
Error occurred at line 64 in procedure writelastname
Last called from line 90 in procedure buildcustomerfile
Last called from line 103 in program customs
```

The first line of the walkback contains header information about the program and gives the type of error (fatal or I/O) and the program counter at the time of the error.

The second line of the walkback is the error message issued by the error reporting system. Appendix B of the Programmer’s Guide contains a list of run-time errors and a short explanation of what may have gone wrong. (If the error is I/O-related, an additional line is printed that provides the system I/O error code and the name of the file causing the error.)

The third line shows the location of the error in terms of source program lines. The remaining lines of the walkback indicate the reverse order in which the procedures were called.

Pascal-2 also includes the capability to trap and recover from run-time I/O errors and to print additional information about the error. See the Programmer’s Guide for discussion of these more advanced techniques.

Handwritten text, likely a letter or document, covering the upper portion of the page. The text is faint and mostly illegible due to fading.

Handwritten text, likely a letter or document, covering the middle portion of the page. The text is faint and mostly illegible due to fading.

Handwritten text, likely a letter or document, covering the lower portion of the page. The text is faint and mostly illegible due to fading.

Compilation Options

The Program Listing

Many times, to correct an error, you need to see more of the program than just the line on which the error appears. The Pascal-2 compiler can be directed to display the entire program, with all detected errors and other information. This is the "listing" of the program.

To obtain a listing file (.LST), include the `list` switch in the compilation command line:

```
.R PASCAL  
*SECOND/LIST
```

To get a program listing at a terminal, specify `TT:` as the listing file, as shown below. The listing also may be written to the line printer or a disk file.

```
.R PASCAL  
*THIRD,TT: = THIRD/LIST  
Pascal-2 RT11 SJ V2.1A 5-Aug-83 7:04 PM Site #1-1 Page 1-1  
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760  
THIRD,TT: = THIRD/LIST
```

```
1      program Third (output)
                                     ~19
*** 19: Use ';' to separate statements
2      begin
3          writeln ('Things get hazy if you stare at them');
4      end.
```

```
***There was 1 line with errors detected ***
```

The listing is printed in pages, with a heading on each page showing the program name, the exact version of the Pascal-2 compiler, the date and time, and the licensed user identification. The listing also prints out, in the left-hand column, the line number for each line of the program. You also may use the `errors` switch to create a listing file containing only the lines with detected errors.

As illustrated in the above example of `list`, a compilation switch modifies the compilation process in some way. A switch is signified by a slash and a descriptive name. The Programmer's Guide describes all of the compilation switches, but the next examples show the most commonly used ones. The examples also demonstrate some of the features of the Pascal-2 package — the Debugger, the Profiler, and the PASMAT formatter, and double-precision real number format.

The Formatter

Suppose you have a program, `EFACT.PAS`, that calculates an approximation of e , the base of the natural logarithms, by summing the series

$$1 + 1/1! + 1/2! + 1/3! + \dots + 1/N!$$

until additional terms do not affect the approximation.

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the above matter.

I am sorry to hear that you are not satisfied with the result of the examination. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible. I have been very anxious to see that all the necessary precautions were taken, and I am sure that the result was as fair as possible.

Remember that the compiler will accept a program in whatever format you choose. So the program may look like this:

```

program Efact(output);
var E, Delta, Fact: real;
N: integer;
begin
E:=1.0; N:=1; Fact:=1.0; Delta:=1.0;
repeat
E:=E+Delta;
N:=N+1; Fact:=Fact*N; Delta:=1/Fact;
until E = (E+Delta);
write('With ', n:1, ' terms, ');
writeln('the value of e is',E:18:15);
end.

```

To make the program more readable, you decide to format the program with PASMAT, one of the Pascal-2 utility programs. Give the following command:

```

.R PASMAT
*EFACT

```

and the program is reformatted to look like this:

```

program Efact(output);

var
    E, Delta, Fact: real;
    N: integer;

begin
    E := 1.0;
    N := 1;
    Fact := 1.0;
    Delta := 1.0;
    repeat
        E := E + Delta;
        N := N + 1;
        Fact := Fact * N;
        Delta := 1 / Fact;
    until E = (E + Delta);
    write('With ', n:1, ' terms, ');
    writeln('the value of e is',E:18:15);
end.

```

(PASMAT has many other formatting options. See the Utilities Guide for details.) Now proceed to compile the program.

```

.R PASCAL
*EFACT

```

```

.LINK EFACT,SY:PASCAL
.RUN EFACT

```

With 11 terms, the value of e is 2.7182820000000000

The Debugger

Even after you have corrected any syntax errors caught by the compiler, the program may still yield unexpected results. In this situation, Pascal-2's interactive Debugger can help uncover and correct the problems. The Debugger takes control of the program and responds to your commands, displaying execution information in a Pascal context. With the Debugger, you can watch the progress of the computation, and you can display intermediate values without making any program changes. You can then spot the point at which values go awry and correct the error.

To do this, use the `debug` switch to compile the program with the Debugger. (In most cases, you probably also will want to overlay the Debugger module. See the Debugger Guide for details.)

First, compile and link the program with the commands:

```
.R PASCAL  
*EFACT/DEBUG
```

```
.LINK EFACT,SY:PASCAL
```

The `debug` compilation produces four output files: `EFACT.LST`, `EFACT.SYM`, `EFACT.SMP`, and `EFACT.OBJ`. You need the listing file to determine the places to set breakpoints in the program. Don't worry about the other three output files, but don't delete them or the listing file. The Debugger uses all of them.

After doing a `debug` compilation, you will find it handy to have a printout of the listing file. The file will look like this:

```
Pascal-2 RT11 SJ V2.1A   5-Aug-83   7:04 PM   Site #1-1   Page 1-1  
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760  
EFACT/DEBUG
```

```
Line Stmt  
  1      program Efact(output);  
  2  
  3      var  
  4          E, Delta, Fact: real;  
  5          N: integer;  
  6  
  7  1  begin  
  8  2      E := 1.0;  
  9  3      N := 1;  
 10  4      Fact := 1.0;  
 11  5      Delta := 1.0;  
 12  6      repeat  
 13  7          E := E + Delta;  
 14  8          N := N + 1;  
 15  9          Fact := Fact * N;  
 16 10          Delta := 1 / Fact;  
 17      until E = (E + Delta);  
 18 11      write('With ', n:1, ' terms, ');  
 19 12      writeln('the value of e is',E:18:15);  
 20      end.
```

```
*** No lines with errors detected ***
```

Two columns of numbers appear on the left side of each page. The first column, labeled `Line`, numbers each line of the source program. The second column, labeled `Stmt`, gives the statement

number of the first statement on that line. The statement numbers start at 1 for each procedure or function, increasing by one as each statement is compiled. The Debugger uses these statement numbers to identify breakpoint locations in the compiled program.

In the program `Efact`, for instance, you may want to set a breakpoint at statement number 7. This is the point at which the approximation of `e` changes. If the program compiles correctly but produces unsatisfactory results, you may set the breakpoint at `MAIN,7` to monitor the approximation to `e` as the program runs. We'll do just that in the next example.

Notice that the Debugger allows you to set the breakpoints. In this example, you tell the program to write the value of `e` at the breakpoint and then continue. (See the Debugger Guide for details on these commands.)

.RUN EFACT

Pascal Debugger V3.00 -- 12-Aug-83

Debugging program: EFACT

} B(MAIN,7) <W(E);C> _____ at breakpoint, write E and continue
 } G _____ start program

```
Breakpoint at MAIN,7  E := E + Delta;
1.0000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.0000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.5000000E+00
Breakpoint at MAIN,7  E := E + Delta;
2.6666667E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7083335E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7166669E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7180557E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182541E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182789E+00
Breakpoint at MAIN,7  E := E + Delta;
2.7182817E+00
With 11 terms the value of e is 2.718282000000000
```

Program terminated.

Breakpoint at MAIN,12 writeln('the value of e is', E: 18: 15);
 } Q _____ quit

Double Precision

The computed value in the previous examples is printed with 7 significant digits. You may need greater precision for some programs. To get extended precision, use the **double** switch, which computes to 15 significant digits. The **double** switch allows you to print a more precise value for **e**. (See the Programmer's Guide for details.)

```
.R PASCAL  
*EFACT/DOUBLE
```

```
.LINK EFACT,SY:PASCAL  
.RUN EFACT
```

With 19 terms, the value of **e** is 2.718281828459050

The Profiler

Finally, let's examine the program for efficiency by using the **profile** switch and by adding the **PRFILE** module to the Linker input. "Profiling" shows the number of times each statement is executed, giving you the opportunity to optimize sections of code that are executed many times.

The Profiler takes control of your program and asks for the name of the profile output file. The default extension is **.PRO**.

```
.R PASCAL  
*EFACT/PROFILE
```

```
.LINK EFACT,SY:PRFILE,SY:PASCAL  
.RUN EFACT
```

profile V2.1 12-Aug-83

Profiling program: EFACT

Profile output file name? EFACT ——— Output goes to default extension
With 11 terms, the value of **e** is 2.7182820000000000

Program terminated.

Profile being generated

The first part of the document is a letter from the President of the United States to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The letter is written in a very formal and dignified style, and it is full of references to the Constitution and the principles of the Republic. It is a very good example of the style of the time.

The letter is also very interesting, as it shows the President's views on the issues of the day, and it shows how he was trying to deal with the problems of the new administration.

The letter is a very important document, and it is one of the most important documents in the history of the United States. It is a very good example of the style of the time, and it is a very good example of the President's views on the issues of the day.

The letter is also very interesting, as it shows the President's views on the issues of the day, and it shows how he was trying to deal with the problems of the new administration.

The letter is a very important document, and it is one of the most important documents in the history of the United States. It is a very good example of the style of the time, and it is a very good example of the President's views on the issues of the day.

The output file looks like this:

Pascal-2 RT11 SJ V2.1A 5-Aug-83 7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
EFACT/PROFILE

```

Line Stmt
1      program Efact(output);
2      var
3          E, Delta, Fact: real;
4          N: integer;
5
1  6      1  begin
1  7      2  E := 1.0;
1  8      3  N := 1;
1  9      4  Fact := 1.0;
1 10     5  Delta := 1.0;
10 11     6  repeat
10 12     7      E := E + Delta;
10 13     8      N := N + 1;
10 14     9      Fact := Fact * N;
10 15    10      Delta := 1 / Fact;
16      until E = (E + Delta);
1 17    11      write('With ', n: 1, ' terms ');
1 18    12      writeln('the value of e is', e: 18: 15);
19      end.

```

*** No lines with errors detected ***

PROCEDURE EXECUTION SUMMARY

Procedure name	statements	times called	statements executed
MAIN	12	1	57 100.00%

There are 12 statements in 1 procedures in this program.
57 statements were executed during the profile.

The leftmost column of the profile listing shows the number of times each line is executed. The Profiler listing concludes with a "Procedure Execution Summary" that details each procedure name, the number of times it is called, the number of statements it contains, and the number of statements it executes. Note, too, that the summary shows the percent of execution count taken by each program block. (In this example, with only one procedure, the portion is 100%.) Given this information, you can attempt to optimize the procedures and statements that use a disproportionately large part of the time ("90 percent of the time on 10 percent of the program").

See the Profiler section of the Debugger Guide for more information and for a much more detailed example.

Your Next Step

Thus ends your guided tour through Pascal-2. At this point, you should be able to run a few simple programs. Before getting into complex programs, however, you should consult the Programmer's Guide, the Language Specification, and the Debugger Guide.

Handwritten text at the top of the page, possibly a title or header.

Second paragraph of handwritten text.

Third paragraph of handwritten text.

Fourth paragraph of handwritten text.

Fifth paragraph of handwritten text.

Sixth paragraph of handwritten text at the bottom of the page.

Pascal-2 V2.1/RT-11 Programmer's Guide

Introduction

The Programmer's Guide contains nitty-gritty information about Pascal-2 for programmers well-versed in the Pascal language. This guide describes compiler commands, compilation and embedded switches, I/O control switches, and Pascal-2's low-level interaction with the PDP-11. This guide also describes ways to handle common Pascal-related implementation questions on RT-11 and contains other miscellaneous information.

This guide is not:

- an introduction to Pascal (see *Programming in Pascal* by Peter Grogono);
- a beginner's guide to Pascal-2 (see the User Guide);
- a detailed description of Pascal-2 (see the Pascal-2 Language Specification).

Compiler Commands

All Pascal-2 compilation commands are divided into three parts: the compiler invocation command, the file specifications, and the compilation switches.

The compilation syntax for Pascal-2 is this:

```
.R PASCAL  
*output-file,listing-file=input-files/switches
```

The **R PASCAL** invocation (or some other name that your system manager has chosen for the invocation command) must always come first. The '*' prompt appears for the file specifications and compilation switches.

input-files:

The only required file specification is at least one input file. Multiple input files are concatenated in order, from left to right, so that a large program can be split into separate files or so that a common set of definitions can be placed in a configuration file. With "source concatenation" no input file can contain a program statement, except for the first file listed. If no output specification is given, the output is determined by the compilation switches; the file name is taken from the last input file specified; and the output files will be placed in the default directory. The default input file extension is .PAS. Multiple input files are separated by a comma.

output-file:

The output file specifies the name of the object output, with a default extension of .OBJ. If the **macro** compilation switch is specified, the output file contains MACRO-11 code and the default extension is .MAC.

listing-file:

The listing file specifies the file to receive the compilation or error listing. The default listing file extension is .LST.

If an equal sign appears on the command line, but no file name is listed in the position of the output file, no output file is generated. If no file name is listed in the

of the

... ..

... ..

... ..

... ..

... ..

... ..

... ..

... ..

position of the listing file, a listing output is produced only if errors exist; if errors exist, output is sent to the user's terminal with the errors switch assumed.

switches: Program compilation is affected in some way by one or more of the options described in the next section. Examples in this manual show the compilation switches after the last file specification, but switches may appear after any file specification and wherever they're placed, they apply to the entire compilation. Multiple switches are separated by slashes.

Compilation Switches

Compilation switches provide control over the files generated and over some aspects of the generated code. A switch is signified by a descriptive name (e.g., **check**). A switch name beginning with **no** reverses the effect of the switch (e.g., **nocheck**). A switch name may be abbreviated as long as the shortened form is sufficient to identify the switch. Three characters of the switch name (excluding the **no**) always identifies a Pascal-2 compilation switch (e.g., **che**, **noche**; **mac**, **nomac**).

Some switches, such as **object** and **macro**, are incompatible, causing the error message "conflicting switches specified" if used in the same compilation.

Pascal-2 compilation switches are:

Program Options

- double** All real variables are in 8-byte floating-point format. You also must use colon notation (e.g., **E:18:15**) within the program to obtain double-precision values in the **write** statement. Default is "off": real variables are in 4-byte format. See "Extended Precision" for more details.
- pascal1** Specifies that the interface to external procedures be compatible with Pascal-1. This interface is a bit less efficient than that of Pascal-2; the **pascal1** switch should be used only when required. Default is the Pascal-2 interface.
- nomain** No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is **main**: a main program is being compiled.
- own** Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with **own**. Default is "off": global variables are shared.

nowalkback

Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. Default is **walkback**: the tables are generated, and the full walkback in source terms is displayed after the message header and error message. See "Run-Time Error Reporting" later in this guide for a discussion of the walkback. The debug switch disables the generation of the error walkback.

Compiler Options

workspace:n

By default, the Pascal-2 compiler opens about 500 blocks of work space. The **workspace** option allows you to reduce work space to compile a small program. A realistic low range is about 150 to 200 blocks. The error message "attempt to write past eof"

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

on a work file indicates that the compiler needs more work space for a large program. A realistic high range is about 750 blocks.

- errors** Requests that the listing file contain only lines with errors. Unless **list** is specified, the default is "on" and errors are printed on the terminal. When **list** is specified, the default is "off" and all lines are printed in the listing file. This switch has no effect when used with the **debug** switch or the **profile** switch, because both of these switches always generate a listing file.
- list** Requests a full source listing in the listing file. If a listing file is specified, the default is "on"; otherwise the default is "off."
- debug** Requests generation of code and auxiliary files to interface with the Pascal-2 Debugger. Default is "off." The **debug** switch disables the generation of the walkback. This switch cannot be used with the **profile** switch or the **errors** switch.
- profile** Requests an execution profile when the program is run. Default is "off." The switch cannot be used with the **debug** switch or the **errors** switch.

Code Switches

- object** Generates an object format output file with default extension .OBJ. Default is normally "on"; object code will be generated. The switch is "off" when **noobject** is specified or when no output file is provided on the command line. The switch cannot be used with the **macro** switch.
- macro** Generates MACRO-11 code in the output file. This code may be assembled by the **MACRO** assembler command to produce an object file. When **macro** is specified, **object** is set "off" and the default extension for the output file becomes .MAC. Default of **macro** is "off." The **macro** switch cannot be used with the **object** switch.

Checking Switches

- nocheck** Disables all run-time checks, including index range checks, subrange assignment checks, pointer checks, stack checks, case label checks, and divide-by-zero checks. Note that compilation errors are still detected. Thus, if **nocheck** is specified, **var A:array [2..10] of integer; A[1] := 0;** will still be detected as a compilation error, but **I := 1; A[I] := 0;** will not be. After a program has been fully debugged, the **nocheck** switch can be used to reduce the size of the compiled code. Default is **check**.
- standard** Requests that all Pascal-2 extended language features be flagged as errors. Default is **nostandard**.
- test** Used in debugging the compiler. Default is "off."
- times** Prints wall-clock time consumed by the compiler and the compilation rate in lines per minute. Default is "off."

Processor Switches

The processor switch defaults to the processor option for the machine on which the compiler is running. Change the value by specifying exactly one of these four switches on the command line:

- fpp** Requests the compiler to generate code for a machine with the Floating Point Processor (FPP) option. FPP instructions include **ADDF**, **MODF**, **DIVF**, etc. This switch implies the

1870

My dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the matter of the

and in reply to inform you that the same has been forwarded to the proper authorities for their consideration.

I am, Sir, very respectfully,
Your obedient servant,
J. H. [Name]

Pascal-2 V2.1/RT-11 Programmer's Guide

eis switch and may not be specified at the same time as the **fis** switch.

fis Requests the compiler to generate code for a machine with the Floating Instruction Set (FIS) option. FIS supports only the four basic floating-point instructions and is available on only a few types of machines. This switch implies the **eis** switch and may not be specified at the same time as the **fpp** switch.

eis Requests the compiler to generate code for a machine with the Extended Instruction Set (EIS) option. The EIS processor option includes instructions to perform integer multiplication and division. Floating-point operations will be done with calls to a floating-point simulator.

sim Requests code with calls to software routines for integer multiply and divide as well as for floating-point arithmetic. Should be used only if the target machine does not have EIS.

Embedded Switches

Some characteristics of the compiled code may be controlled by switches included in the source code. These switches take the form of a Pascal comment beginning with a dollar sign '\$' and followed by a descriptive name, for example:

```
{ $indexcheck }
```

A switch name beginning with "no" reverses the effect of the switch, for example:

```
{ $noindexcheck }
```

Most switches may be abbreviated to a minimum of three characters, for example:

```
{ $ind } or { $noi }
```

However, when using **\$nopointercheck** and **\$nopprofile** be sure to enter more than three characters, or the compiler will treat the switch as an ordinary comment.

Multiple switches can be embedded within a single comment. The switches must be separated by commas; only the first may have the dollar sign. The following forms are equivalent:

```
{ $noindex, norange }  
{ $noindex } { $norange }
```

Embedded switches are counting switches. Each occurrence increments or decrements the switch value; the switch is enabled if its value is greater than zero. The initial value of a switch is controlled by an equivalent compilation switch, such as **debug**, if the equivalent compilation switch exists. If no equivalent switch is present on the command line, the initial value is determined by the defaults described below.

Once set, some switches are valid for the entire program, as with **\$own**. In some cases, the "no" form of the switch is the one normally used, as with **\$nomain**.

Some switches may be turned "on" and "off" for a particular section of code, either on a statement-by-statement or procedure-by-procedure basis. The following example shows how debugging can be

turned off for a procedure:

```

:
{ $nodebug } ----- debugging turned off

procedure P;
begin
: ----- body of procedure P
end;

{ $debug } ----- debugging enabled again
:

```

The particulars of each switch are described in the following sections.

Program Options

\$double Specifies that all real arithmetic is to be done with double precision rather than with single precision. **\$double** applies to the entire compilation. You also must use colon notation (e.g., E:18:15) to print the double-precision values in a **write** statement. This switch must appear in the program before any data of type **real** is defined or used. Default is "off."

\$pascal1

Specifies that external procedures will be called in a manner compatible with Pascal-1. This switch may slow program execution but should simplify conversion of programs from Pascal-1 to Pascal-2. The default is "off."

External Pascal-2 procedures may be called regardless of the setting of this switch.

\$nomain No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is **\$main**: a main program is being compiled.

\$own Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with **\$own**. **\$Own** applies to the entire compilation. Default is "off": global variables are shared.

\$nowalkback

Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. Default is **walkback**: the tables will be generated, and the full walkback in source terms will be displayed following the message header and error message. The **debug** compilation switch disables the generation of the error walkback. See "Run-Time Error Reporting" later in this guide for a discussion of the walkback.

Compiler Options

\$nodebug, \$debug

Disables/enables some of the overhead of the Pascal-2 Debugger. These two switches will have effect only when the **debug** compilation switch is specified. The **debug** switch generates the extra files needed for debugging and sets the **\$debug** switch "on."

\$Nodebug can be used to turn off some of the debugging overhead for procedures or functions that have already been fully tested. **\$Debug** can be used to restore debugging for other procedures.

\$noprofile, \$profile

Disables/enables some of the overhead of the Pascal-2 Profiler. These two switches will have effect only when the **profile** compilation switch is specified. The **profile** switch generates the extra files needed for profiling and sets **\$profile** "on." **\$Noprofile** can be used to turn off profiling for procedures or functions that do not need to be profiled, and **\$profile** can be used to restore profiling for other procedures.

When the **begin** statement of a procedure is compiled, the state of the **\$debug/\$nodebug** and **\$profile/\$noprofile** switches will determine debugging or profiling for that entire procedure. Note that a procedure constitutes the smallest section of code that can be debugged or profiled; you can't debug or profile individual lines of a procedure.

The **\$debug/\$nodebug** and **\$profile/\$noprofile** switches serve the same functions as far as the code generated. You would never use both sets in the same compilation. (You can't debug the program and profile it at the same time.)

\$nolist Turns off the listing of source lines in the listing file; **\$list** restores the listing of source lines. The switch may be turned on or off after each line of source code. The listing file will display the **\$nolist/\$list** switches, and the line numbers will reflect the lines for which listing has been disabled. In this program fragment, listing has been disabled on lines 3 through 5:

```
1      program Ex(output);
2      {$nolist}
3      {$list}
4
5      begin
6      :
```

Lines with errors will be displayed even if the **\$nolist** switch is on. Default is **\$list**.

Do not use the **\$nolist** switch during debugging sessions. If you attempt to access any "unlisted" line(s), the response will be the message "No such statement in this procedure." Other errors also may be produced.

\$standard

Like the corresponding compilation switch, **\$standard** causes all extended language features of Pascal-2 to be flagged as compilation errors. By using the embedded switch at the beginning of the program, you don't have to use the **standard** switch every time you compile the program.

In addition, if you want to compile the program using language extensions of Pascal-2, but you want to mark the non-standard features (for later transportability to another compiler, perhaps), insert the **\$standard** switch at the start of the program, and enclose any non-standard sections with the switches **\$nostandard** and **\$standard**. The compiler will then check the rest of the program for non-standard features, so that you can minimize your use of extensions. The **\$nostandard** switch will be a textual flag to aid any future conversion to a standard program.

The **\$standard** and **\$nostandard** switches may be turned on or off after each line of source code. Default is **\$nostandard**, which accepts the extended language features of Pascal-2 as correct forms.

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the matter of the

above named case, and in reply to inform you that the same has been forwarded to the proper authorities for their consideration.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

Very respectfully,
Your obedient servant,

J. H. [Signature]

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

Run-Time Checking Switches

The compilation switch `nocheck` will turn off all run-time checks. The embedded switches listed below will cancel the particular checks listed below. Any of these switches can be placed at the start of the program to turn off a particular kind of check throughout. Or, "on/off" pairings can be used on a statement-by-statement basis within the program.

Turning off run-time checks will reduce the size of the program. However, we recommend that you do not turn off any checks until the program has been fully debugged.

`$noindexcheck`

Stops generation of code for array bounds checks; no array index is checked as to whether it is within the array bounds. Default is `$indexcheck`.

`$nopointercheck`

Stops generation of code that checks for nil or invalid pointer values. Default is `$pointercheck`.

`$norangecheck`

Cancels the subrange assignment and `case` statement check capabilities. No assignment to a variable of subrange type is checked as to whether the assigned value is within the allowed range. Also, `case` selectors are not checked for matching labels. Default is `$rangecheck`.

`$nostackcheck`

Stops the generation of code for stack overflow checks on procedure and function entry. No entry to a procedure or function is checked as to whether adequate stack space is available for local variables. Note that some procedures call support library routines that check for stack overflow. Thus, even when compiled with this switch, some programs may still report "stack overflow" errors. The default is `$stackcheck`.

Compilation Examples

The following examples show the effects of various switches on the compilation.

Example 1.

```
.R PASCAL
*PROG/LIST
```

Compiles the file `PROG.PAS` and generates an object file `PROG.OBJ` and a listing file `PROG.LST`. The `check` switch is assumed to be on, and code will be generated for the hardware options of the machine on which the program is being compiled.

Example 2.

```
.R PASCAL
*PROG,PROG=PROG
```

Equivalent to Example 1.

Example 3.

```
.R PASCAL
*PROG=PROG/NOCHECK/FIS
```

Compiles the file `PROG.PAS` and generates an object file `PROG.OBJ`. Any errors will be listed on the user's terminal. No run-time checking code is generated, and code will be generated for a CPU with FIS instructions.

1. The first part of the paper discusses the importance of the study of the history of the United States. It is argued that a knowledge of the past is essential for a full understanding of the present and for the development of a sound policy for the future.

2. The second part of the paper discusses the role of the government in the development of the United States. It is argued that the government has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

3. The third part of the paper discusses the role of the individual in the development of the United States. It is argued that the individual has played a crucial role in the development of the country, and that his actions have been guided by a set of principles that have been developed over time.

4. The fourth part of the paper discusses the role of the community in the development of the United States. It is argued that the community has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

5. The fifth part of the paper discusses the role of the nation in the development of the United States. It is argued that the nation has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

6. The sixth part of the paper discusses the role of the world in the development of the United States. It is argued that the world has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

7. The seventh part of the paper discusses the role of the future in the development of the United States. It is argued that the future has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

8. The eighth part of the paper discusses the role of the past in the development of the United States. It is argued that the past has played a crucial role in the development of the country, and that its actions have been guided by a set of principles that have been developed over time.

Pascal-2 V2.1/RT-11 Programmer's Guide

Example 4.

```
.R PASCAL  
*HEADER,MIDDLE,PROCED/NOMAIN
```

Concatenates and compiles the files HEADER.PAS, MIDDLE.PAS, and PROCED.PAS in the order given, and generates an object file, PROCED.OBJ. This code has no main body and therefore contains external procedures. The **check** switch is assumed to be on, and code will be generated for the hardware options of the machine on which the program is being compiled.

Example 5.

```
.R PASCAL  
* ,TT:=PROG
```

Produces a listing file to the terminal but no PROG.OBJ file.

Linking and Executing

After compilation, Pascal-2 programs must be linked to the support library before being executed. The DCL sequence is:

```
.LINK PROG,SY:PASCAL  
.RUN PROG
```

The preceding two commands may also be entered in following manner as well:

```
.R LINK  
*PROG = PROG,SY:PASCAL  
*~C  
.RUN PROG
```

See "The Linker, Overlays, and the Librarian" for details of the link process.

I/O Control Switches

The **reset** and **rewrite** standard procedures accept two additional arguments specifying the file name of an external file and default fields of the file name. These arguments can also include I/O control switches that give explicit control of the operating system interface details. (A fourth parameter can also be specified. See the Language Specification for a complete discussion of **reset** and **rewrite**.)

The I/O switches appear in the file name or default name parameters as shown in these examples:

```
rewrite(F, 'data.dat/seek/span/size:12.', ErrStatus);
```

A special device (TI:) also may appear in the **reset** and **rewrite** calls. The TI: device connects to the Pascal-2 terminal driver and is used in place of the TT: driver for interactive use.

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters. The parameter *n* is a octal number unless terminated by a period, in which case it is a decimal number.

<u>Switch</u>	<u>Meaning</u>
/buffersize:n	Allocate <i>n</i> bytes for buffer (hereafter abbreviated /buff)
/go	Allow programmed error handling
/nfs	Non-File-Structured access
/odt	Single-character terminal input
/seek	Direct-access file
/size:n	File storage allocation
/span	Records span block boundaries
/temp	Temporary file

/buff:n (Buffersize): The **/buff:n** switch specifies the storage to be allocated to a file buffer. Pascal-2 normally allocates the minimum space required for a file buffer, 512 bytes. For disk files this default value may be raised by multiples of 512 to improve the efficiency of I/O transfers, at the cost of additional memory. Line-oriented files such as terminal and line-printer files receive buffer space equal to the width of the line, usually 80 characters. Efficient single-character I/O requires a minimum of buffer space, 1 or 2 characters. See also the **/odt** switch below.

The value of *n* used with **/buff:n** is a decimal number if terminated with a period; otherwise octal.

/go (I/O Error Continue): I/O transfer errors are normally fatal and cause immediate program termination. The **/go** switch indicates that I/O errors on the specified file are non-fatal and allow the program to continue executing. This switch is the equivalent of **noioerror**, one of three predefined Pascal procedures that trap and respond to I/O errors. See the section "I/O Error Trapping." The switch **/go** has been left in for compatibility with previous implementations.

/nfs (Non-File-Structured Access): The **/nfs** switch is used to achieve non-file-structured access to a device that is normally file-structured, such as a disk device. Because direct access to such a device can destroy its directory structure, Pascal-2 prevents non-file-structured access unless the **/nfs** switch is used.

/odt (Single-Character I/O): The **/odt** switch derives its name from the ODT Debugger (Octal Debugging Technique), which is driven by single-character commands. The **/odt** switch is used with keyboard files, indicating that each character read from the file is to be processed immediately without any wait for a carriage return or other action

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE
THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE
THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE
THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE
THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE
THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

Pascal-2 V2.1/RT-11 Programmer's Guide

character. The `/odt` switch is only effective for files on the `TI:` terminal device, and the size of the buffer should be reduced to a minimum, as shown:

```
reset(input, 'ti:/odt/buff:2');
```

Note that the RUBOUT and Control-U (~U) keyboard editing capabilities are not effective with `/odt`.

- `/seek`** (Direct-Access): The `/seek` switch enables the use of the direct-access `seek` procedure, and it permits both read and write access to the file variable so that records may be updated with calls to `get` and `put`. The `seek` procedure is described in the Language Specification.
- `/size:n`** (File Allocation): The `/size:n` switch used in the `rewrite` procedure specifies the space to be allocated for the file. The size of the file is given in blocks of 512 bytes.
- `/span`** (Spanned): In files created or accessed by Pascal-2 programs, fixed-length records are normally "blocked." This means that an integral number of records are stored in one disk block of 512 bytes, with any remaining storage in that block being unused. The `/span` switch packs records more efficiently, with records spanning from one disk block to the next. This requires additional buffer memory, which is automatically allocated, and some additional computation. Spanned and blocked files are not generally compatible. Files created with `/span` should be read with the same switch.
- `/temp`** (Temporary): This switch is used with `rewrite` to indicate a temporary file that will be deleted on termination. No file name is needed if this switch appears.

Handwritten text, likely bleed-through from the reverse side of the page. The text is mostly illegible due to fading and the quality of the scan. It appears to be a letter or a document with several paragraphs.

External Modules

Pascal-2 implements separate compilation through the concept of an external module, a program fragment containing at least one procedure or function. External modules are compiled independently of other program units and combined by the Linker. External modules may be stored in libraries to simplify the handling of common routines.

External modules may reference global variables shared by all of the modules making up a program. If each module (including the main program) is compiled with the same global variables, the effect is as if all modules were compiled together. For this to work properly, the global declarations in the external procedure and main program must contain the same variable declarations in the same order. Parameter lists also must agree (i.e., contain the same parameter declarations in the same order). The simplest and most efficient way to do this is to place all global declarations, including references to external procedures and functions, in a header file then include the header file in the external module and in the main program, using the `%include` compiler directive (see "Multiple Source Files").

External modules must be referenced at the outermost (global) level of a main program, but they may be called from any point in the code. An external module compilation requires either the `nomain` compilation switch or the `$nomain` embedded switch. Both switches specify that no main program is contained in the source file. The `nomain` switch is specified on the command line, whereas the `$nomain` embedded switch is placed at the beginning of the external module.

An external procedure name consists of the first six characters of the procedure or function identifier. External procedure names must uniquely identify an external routine because they are used as global symbol names by the Linker.

Linking errors most pertinent to external modules are:

- Duplication of external symbols. An external procedure has been defined in more than one module. When multiple definitions are encountered, the Linker uses the first definition only and ignores succeeding definitions.
- Undefined external symbols. The program has referenced an external routine that was not defined in any of the object files or libraries specified on the command line. This error indicates that the program contains unresolved global references.

In each case, the Linker responds with an error message and produces an output file that cannot execute (permission bits on the file are not set).

Two compiler directives, `external` and `nonpascal`, allow the use of external modules. The `external` directive defines a procedure or function implemented in Pascal-2 as "external," which means that the procedure may be referenced by other modules and that both the external module and the program or module that calls it expects to find the normal Pascal-2 calling sequence of parameters on the stack. The `nonpascal` directive defines a routine written in a language other than Pascal, such as FORTRAN or MACRO-11, and generates a call to the FORTRAN interface routine (P\$111) in the Pascal support library. P\$111 creates the standard DEC calling sequence of parameters which is expected by the external module, and which differs from Pascal-2's.

CAUTION

Observe two cautions when using the `external` or `nonpascal` directive. Parameters to external routines cannot be checked by the compiler for type conformance across module boundaries, so an accidental type mismatch may cause unpredictable results. Also, the compiler cannot verify the conformance of global data. As mentioned above, use of the `%include` directive can help reduce problems in this area.

The first of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The second of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The third of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The fourth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The fifth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The sixth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The seventh of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The eighth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The ninth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The tenth of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

The eleventh of these is the fact that the system is not a simple one, but a complex one, involving many different factors, and it is not possible to give a simple answer to the question of what is the best system to use.

Calls to Pascal-3 Routines

The syntax of the **external** directive is similar to the syntax of the **forward** directive in that it consists of two distinct parts: the declaration and the body. The declaration includes the external procedure or function name and the argument list, followed by the **external** directive.

```
procedure GetString(Arg: Argtype); _____ this is the declaration
external; _____ external directive is required
```

This declaration must appear in the external module and in each compilation unit that calls that external routine. You can place the declaration in a header file and use the **%include** directive to insert it appropriately.

The body of the external module contains the actual code for the procedure or function and must not include an argument list.

```
procedure GetString; _____ this is the procedure body
begin
:
end;
```

The body and declaration must be compiled together in order for the external procedure to function properly. The external procedure may then be called in the same way as any other procedure or function:

```
GetString(Length); _____ external procedure call
```

Example Using External Directive

The practical application of external procedures is best shown by example. The following sample illustrates the declaration and use of external procedures and the correct way to access global variables. Note that the global declarations must be identical in the external procedure (CHANGE.PAS) and in the main program (MAINLINE.PAS). Note also the use of the **\$nomain** embedded switch in the external procedure.

First, we create a separate header module HDR.PAS containing the external procedure reference and the program's global declarations.

HDR.PAS

```
type _____ global declarations
  GlobalType = record
    B: boolean;
    V: integer;
  end;
var
  Glob: GlobalType;
  I: integer;

procedure Change(P: integer);
external; _____ external directive must appear
```

The file CHANGE.PAS consists of the external procedure **Change** and the **%include** directive, as follows:

Section 1

The first part of the document discusses the importance of maintaining accurate records. It states that without proper documentation, it is difficult to track progress and identify areas for improvement. The text emphasizes the need for a systematic approach to data collection and analysis.

In the second part, the author describes the methodology used for the study. This includes details about the sample size, data sources, and the statistical techniques employed. The goal is to ensure that the findings are based on reliable and valid information.

The third section presents the results of the study. It shows that there is a significant correlation between the variables being studied. The data suggests that the proposed model is effective in predicting the outcomes of interest.

Finally, the document concludes with a summary of the findings and some recommendations for future research. The author suggests that further studies should be conducted to explore the underlying mechanisms and to test the model in different contexts.

The author expresses gratitude to the participants and the funding agency for their support throughout the project.

References are provided at the end of the document, listing the key sources used in the research.

The document is signed by the author, who is a senior researcher in the field.

CHANGE.PAS

```

{$nomain} _____ embedded switch
#include hdr.pas _____ header module
procedure Change; _____ no parameters here
begin
  with Glob do
    begin _____ change global variables
      B := true;
      V := V + P;
    end;
end;

```

The external procedure is then compiled with the `nomain` option embedded, using the command:

```

.R PASCAL
*CHANGE

```

However, you can omit the embedded option and specify the `nomain` on the command line in the following manner:

```

.R PASCAL
*CHANGE/NOMAIN

```

The externally defined procedure `Change` may now be called within any program unit that includes `HDR.PAS` and is linked with `CHANGE.OBJ`. For example, assume that the file `MAINLINE.PAS` consists of this:

MAINLINE.PAS

```

program Mainline;
#include hdr; _____ external procedure and global declarations

procedure Before;

begin
  with Glob do
    writeln('Before executing Change, B = ',B,' and V = ',V:2,'. ');
end;

procedure After;

begin
  with Glob do
    writeln('After executing Change, B = ',B,' and V = ',V:2,'. ');
end;

begin { program Mainline }
  with Glob do
    begin _____ initialize global variables
      B := false;
      V := 0;
    end;
  I := 45;
  Before;
  Change(I); _____ the external call
  After;
end.

```


Pascal-2 V2.1/RT-11 Programmer's Guide

Compile MAIN as any other main program and link the main program and external module, as shown:

```
.R PASCAL  
*MAIN  
.R LINK  
*MAIL MAIN=MAIL.CHANGE.SY:PASCAL
```

Running MAIN yields these results

```
.RUN MAIN
```

Before executing Change, B = false and V = 0.

After executing Change, B = true and V = 45.

Calls to Non-Pascal-2 Routines

The `nonpascal` directive is used instead of `external` when the external procedure is generated by an assembler or a compiler other than Pascal-2. `Nonpascal` creates an interface between the Pascal-2 calling sequence generated by the main program or module and the standard DEC calling sequence required by FORTRAN and most MACRO routines. In addition, when the `nonpascal` directive is invoked, register R5 is used as a pointer to the list of parameters.

Syntax for the `nonpascal` directive consists of a separate declaration and body. The declaration contains the name of the procedure or function and the argument list, followed by the `nonpascal` directive.

Calling MACRO Subroutines

The external declaration for a MACRO function looks like this:

```
program Test;  
  
var i: integer;  
  
function afunct (var i: integer) : integer;      _____ declaration of  
nonpascal;                                       MACRO routine  
  
begin _____ body of the main program  
  i := 10;  
  writeln(afunct(i));  
end.
```

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

CHICAGO, ILL.

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

CHICAGO, ILL.

THE UNIVERSITY OF CHICAGO

CHICAGO, ILL.

CHICAGO, ILL.

Pascal-2 V2.1/RT-11 Programmer's Guide

The MACRO routine AFUNCT is written:

; Returns argument plus 10

```
AFUNCT:
    mov    02(r5),r0
    add    #12,r0
    rts    pc
    .end
```

Type matching for the declaration and use of parameters are the user's responsibility.

The sample program Test is compiled with the command:

```
.R MACRO
*AFUNCT
.R PASCAL
*TEST
.R LINK
*TEST.TEST=TEST.AFUNCT.SY:PASCAL
.RUN TEST
20
```

MACRO routines written with the Pascal-2 PASMAL utility must be declared as external rather than nonpascal, because PASMAL simulates the Pascal-2 calling sequence.

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

1951

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF THE HISTORY OF ARTS
AND ARCHITECTURE

Calling FORTRAN Subroutines

For Pascal programs to call FORTRAN subroutines, two conditions must be met: all parameters must be passed by reference, not by value, and FORTRAN subroutines must be declared in Pascal programs with the nonpascal directive.

The program FTEST.PAS shows a way to call FORTRAN subroutines from a Pascal program. The program reads three integers from the terminal, calls the FORTRAN subroutine ADDEM to calculate the sum of the three numbers, then prints the sum.

Subroutine ADDEM.FOR contains these statements:

```

SUBROUTINE ADDEM(A,B,C,D)
C
C  ADDS A, B, C TO PRODUCE D.
C
      IMPLICIT INTEGER (A-Z)

      D = A + B + C

      RETURN

END

```

The main Pascal program contains these lines:

```

program FORTest;
var
  A, B, C, D: integer;

  procedure ADDEM(var A, B, C, D: integer);
    nonpascal;

  begin
    write('Enter 3 values: ');
    readln(A, B, C);
    ADDEM(A, B, C, D);  { add the numbers}  ———— FORTRAN call
    writeln('The answer is ', D);
  end.

```

The first part of the report deals with the general situation of the country. It is a very interesting and informative account of the country and its people. The second part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The third part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people.

The fourth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The fifth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The sixth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people.

The seventh part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The eighth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The ninth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people.

The tenth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The eleventh part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people. The twelfth part of the report deals with the specific details of the country and its people. It is a very detailed and informative account of the country and its people.

Pascal-3 V2.1/RT-11 Programmer's Guide

Compile and run the program using these commands. Assume that the FORTRAN subroutine ADDEM has already been compiled.

```
.R PASCAL
•FTEST
.R LINK
•FTEST=FTEST.ADDEM.SY:PASCAL
.RUN FTEST
Enter 3 values: 20 40 120
The answer is 180
```

Two restrictions relate specifically to the use of FORTRAN subroutines in Pascal programs. First, Pascal-called FORTRAN subroutines cannot access files opened in the Pascal program. However, these FORTRAN routines can use files that they themselves open.

Second, FORTRAN allows the passing of "null" parameters to subroutines, in which a comma is used as a place holder for an optional parameter. Pascal has no such feature. To pass null parameters to a FORTRAN subroutine from Pascal, use the `origin` directive to declare the null parameter. The variable and procedure declaration for the Pascal program appears as shown:

```
var
  ListNumber, LastList: integer;
  INull origin 0: integer;  - specifies null integer parameter for FORTRAN
  RNull origin 0: real;    — specifies null real parameter for FORTRAN
  Rewind: boolean;

procedure FPREW(var Number, Last: integer;
               var I: integer; var R: real;
               var Rew: boolean);

nonpascal;
```

The FORTRAN subroutine declaration is then:

```
subroutine FPREW(Number,Last,I,R,Rew)
```

When you call the FORTRAN subroutine from the Pascal program, substitute the appropriate `Null` variable for any unnecessary parameters. In the case of `FPREW`, the third and fourth parameters are null parameters:

```
FPREW(ListNumber,LastList,INull,RNull,Rewind);
```

1. The first part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom.

2. In the second part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

3. In the third part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

4. In the fourth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

5. In the fifth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

6. In the sixth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

7. In the seventh part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

8. In the eighth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

9. In the ninth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

10. In the tenth part, we shall consider the question of the structure of the atom, and the question of the structure of the atom.

External Module Libraries

Suppose you want a library of procedures that can be referenced by any program. For a particular program, you do not necessarily reference all the procedures in that library, and you do not want the entire library loaded with the program.

Procedures and functions from one compilation unit form a single object file and cannot be selectively loaded. For example, if procedures A, B, and C are compiled together and placed in a library, any reference to one of them causes all three to be loaded. On the other hand, if each procedure A, B, and C is compiled separately and the three object files are placed in the same library, then a reference to one of them causes only that procedure to be loaded in the program. To keep final program size to the minimum, library procedures should be compiled separately whenever possible.

Rather than having an external declaration in the main program for each procedure needed, create a single "header" file containing the external declarations for all the external procedures defined in the library. This header file can be included in the compilation with the `%include` directive placed near the beginning of the program source file. No external reference is generated for any external procedure in the header file that is not used by the program, so only those procedures actually used by each compilation unit are loaded into the final image file (assuming that the library procedures were themselves separately compiled). See "Multiple Source Files" for use of the `%include` directive.

By using a header file in this way, you can avoid errors that could be caused by a mismatched declaration, forcing any change made to a declaration for an external procedure to be reflected in all programs using that procedure. Carried to the fullest extent, a library and its corresponding header file can be used system-wide.

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

The Linker, Overlays, and the Librarian

The Linker

Object modules produced by the Pascal-2 compiler are compatible with object modules produced by the MACRO assembler, FORTRAN compiler, and other RT-11 system utility programs. The Linker, LINK.SAV, can produce overlaid executable programs, allowing much larger programs than the 32K-word address space. (Overlays are explained below.) The Librarian, LIBR.SAV, can build libraries of object modules. Some highlights of Linker and Librarian capabilities are covered here. See the *RT-11 System User's Guide* or the *RT-11 System Utilities Manual* for details.

To run the Linker, give the command:

```
.R LINK
```

```
*
```

('*' indicates that the Linker is waiting for a command.)

The first command line can include the output file, a map file if desired, and up to six input files. The file PASCAL.OBJ, which contains the Pascal run-time library, must be used when linking a Pascal program. The example below links the object module MAIN with two others, SUB1 and LIB1, to produce a "privileged" MAIN.SAV file and a MAIN.MAP file. The job is privileged in that the monitor, device handlers and I/O page are mapped into the program's address space, as opposed to being excluded from the program's memory (a "virtual" job, see below).

```
*MAIN,MAIN=MAIN,SUB1,LIB1/C
```

```
*SY:PASCAL
```

```
*~C
```

An alternative to the R LINK command is LINK. LINK, a DCL command, allows you to enter the command line on the same line as the command, as in:

```
.LINK MAIN,SUB1,LIB1,SY:PASCAL
```

Virtual Jobs and the XM Monitor

The XM monitor has the ability to create a "virtual job" with a separate 32K word address space. A virtual job does not need to reserve space for the monitor, device handlers, or the I/O page, and can therefore use the entire available address space. A virtual job cannot make direct reference to device registers or perform other specialized functions. These operations are restricted to "privileged jobs." For most purposes, virtual jobs are preferred.

Making a virtual job requires the setting of bit 10 (2000 octal) in the JSW (Job Status Word) in location 44 (octal) of the .SAV image file. The file VIRJOB.OBJ, supplied with Pascal-2, sets the virtual bit in the JSW for Pascal-2 or other programs. VIRJOB should be included as an input file in the Linker command(s).

```
.R LINK
```

```
*MAIN = MAIN,SUB1,LIB1,VIRJOB/C
```

```
*SY:PASCAL
```

```
*~C
```

One additional restriction: The .SAV image file containing a virtual job must be stored on the system device (SY:), and must be run via the R command.

Overlays

Overlaying is the method of dividing a program logically into small pieces called "segments," allowing you to execute a program that is larger than the available memory (32K). Each overlaid program has a root segment that is always resident in memory and any number of executing overlay segments. Each segment is assigned to a particular area of memory called an overlay region, with each overlay region containing one or more segments.

A program may require (further) overlaying if run-time errors such as "not enough memory" or "stack overflow" cause the program to abort, or when the program cannot be loaded into memory because it is too large.

The Linker is capable of creating two kinds of overlay structures, low memory overlays on the SJ, FB and XM monitors and extended memory or "virtual" overlays on XM. With low memory overlays, only the root and the executing segment reside in the program's 32K address space; all other segments reside on disk and are called in and overlaid against the previous segment, overwriting it. Extended memory overlays require the extended-memory features of the XM monitor to create a physical address space greater than 32K words for the program. This address space is made up of segments assigned to specific virtual overlay regions. Once called into virtual memory to execute, a virtual overlay region remains in physical memory, thus reducing the number of disk accesses to load in the next segment.

In most cases, low memory and extended memory overlays may be mixed for added program flexibility. Details of both types of overlays are given below, oriented toward Pascal tasks. The full overlay capabilities are described in the *RT-11 System Utilities Manual*.

NOTE

The Linker uses channel 15 as the overlay load channel; for this reason, your program should not access channel 15 directly. See the "Support Library" section for information on channels.

Low Memory Overlays

The /O:n (Low Memory Overlay) option selects the low memory overlay facilities of the Linker, where the parameter *n* indicates the overlay region number. Sets of modules allocated to the same region will be overlaid against other modules in the same region, with only one set of modules per region actually in memory at any one time.

The following sequence links a main program and several external modules into an overlaid executable file. The main program and the Pascal library are not overlaid and must be in the root segment (on the first command line). *FIRSTA* and *FIRSTB* do not call each other and are overlaid against each other in region 1. *TWOA* and *TWOB* do not call each other and are overlaid against each other in region 2. The set of *NEXTA*, *NEXT2A*, and *NEXT3A* are overlaid against the set of *NEXTB*, *NEXT2B*, and *NEXT3B* in region 3. No module in one set calls a module in the other set that is overlaid in region 3. The continue option (/C) allows the input file list on the next line to be included in the linking.

```
.R LINK
*PROG = MAIN,SY:PASCAL/C
*FIRSTA/O:1/C
*FIRSTB/O:1/C
*TWOA/O:2/C
*TWOB/O:2/C
*NEXTA,NEXT2A,NEXT3A/O:3/C
*NEXTB,NEXT2B,NEXT3B/O:3
*~C
```

The first part of the report deals with the general situation of the country. It is a very interesting and informative study of the country's development.

The second part of the report deals with the economic situation. It is a very interesting and informative study of the country's economic development.

The third part of the report deals with the social situation. It is a very interesting and informative study of the country's social development.

The fourth part of the report deals with the political situation. It is a very interesting and informative study of the country's political development.

The fifth part of the report deals with the cultural situation. It is a very interesting and informative study of the country's cultural development.

The sixth part of the report deals with the environmental situation. It is a very interesting and informative study of the country's environmental development.

The seventh part of the report deals with the international situation. It is a very interesting and informative study of the country's international development.

The eighth part of the report deals with the future of the country. It is a very interesting and informative study of the country's future development.

The ninth part of the report deals with the conclusion. It is a very interesting and informative study of the country's conclusion.

You can also use the command **LINK/PROMPT**, with the same effect.

Extended Memory Overlays

The **/V:n:m** (Extended Memory Overlay) Linker option is used to create extended memory overlays where *n* is the overlay region number and *m* is the optional partition into which the segment is loaded. Both privileged and virtual jobs can use extended memory overlays. The **/V:n:m** option produces a load image that no longer requires the I/O page and monitor to reside in the program's address space. This makes the entire address space available to the program, an advantage for programs with large root segments or with high demand for dynamic memory (stack and heap).

The next example uses the same program files as in the preceding overlay example to create a virtual job. Note that the **/V:n:m** option replaces the **/O:n** option used above.

In addition to the main program and support library, the root contains the **VIRJOB** code, making the program a virtual job. For comparison, refer to the example in the next section in which the root is a null segment and the main program and support library are loaded into the first (and only) virtual overlay region.

```
.R LINK
*PROG = MAIN,VIRJOB,SY:PASCAL/C
*FIRSTA/V:1/C
*FIRSTB/V:1/C
*TWOA/V:2/C
*TWOB/V:2/C
*NEXTA,NEXT2A,NEXT3A/V:3/C
*NEXTB,NEXT2B,NEXT3B/V:3
*~C
```

Linking a Program Whose Root Exceeds 16K

Due to a restriction in the XM monitor, Pascal programs with root segments or mainlines larger than 16K words cannot be loaded at all. This restriction applies to both privileged and virtual jobs. In the case when overlaying, or further overlaying, of the program is undesirable, use the file **START.OBJ** from the Pascal support library to create a "null" root segment. This places the program code in the first (and only) virtual overlay region. In the process excess memory in the root is added to the heap's free list, increasing the amount of dynamic memory available to the program.

START contains code necessary to create a null root. **START** defines two "grow" psects, **P\$GROW** and **P\$GRWH**, which are used in the creation of the 4K-word null root segment containing low core and vector information (1000₈ words) and **START** (32 words).

The **/U:20000** (Round) Linker option expands **P\$GROW** to the root's upper boundary (**P\$GRWH**). At run time, the Pascal support library places this excess region on the heap's free list for use by the Pascal program.

When **START** is used, any number of segments but only one virtual overlay region can be specified in the overlay structure. Programs having more than one virtual overlay region cannot be linked in this way. As the next example shows, each overlaid module is placed in its own partition within the virtual overlay region. The overlaid module is loaded into virtual memory when first called and resides there for the remainder of the execution, reducing the number of disk accesses. The **V:1:1** option creates virtual overlay region 1, partition 1, containing the tables and code for program **TEST** and the Pascal support library. The **START** and **VIRJOB** code is placed in the root.

Pascal-2 V2.1/RT-11 Programmer's Guide

Example:

```
.R LINK
*TEST = START,VIRJOB/U:20000/C
*TEST,SY:PASCAL/V:1/C
*FIRSTA/V:1:2/C
*FIRSTB/V:1:2/C
*TWOA/V:1:3/C
*TWOB/V:1:3/C
*NEXTA,NEXT2A,NEXT3A/V:1:4/C
*NEXTB,NEXT2B,NEXT3B/V:1:4/C
*//
Round section? P$GROW
*~C
```

For more information on virtual jobs and overlays, see the *RT-11 Software Support Manual* and the *RT-11 System Utilities Manual*.

The Librarian

The Librarian combines relocatable object module files to form an object module library. This library may be included as input to the Linker, which will select only those modules needed by the program being linked. Note that a module always consists of the entire set of procedures and functions from its compilation. Individual procedures cannot be selected from a module.

For example, the dynamic string package STRING.PAS, supplied as a Pascal-2 utility, can be edited to form 12 files, with each file containing one procedure or function. The 12 files can then be compiled and combined into a library containing 12 modules, as follows.

```
.R LIBR
*STRING=LEN,CLEAR,READS,WRITES,CONC,SEARCH/C
*INSERT,ASSIGN,ASCHAR,EQUAL,DELETE,SUBS
*~C
```

Extended Precision

Values of type **real** are normally stored in the PDP-11 single-precision format, which requires 2 words of storage per value and offers about 7 decimal digits of precision. The **double** compilation switch or the **\$double** embedded switch gives double precision to all **real** values. Each extended-precision value occupies 4 words of storage and provides approximately 15-digit precision in all **real** calculations, including the transcendental functions.

Normal and extended-precision values cannot be mixed in a program: the **double** or **\$double** switch generates extended precision for all **real** values. All external modules must be compiled with the same precision as the main program, even if no **real** variables are present.

In addition, you must use the colon notation output format (e.g., E:18:15) to display double precision values in **write** statements.

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

10-17-74 (10-17-74) 10-17-74

Support Library

The Pascal support library is a collection of modules contained in an object module library called PASCAL.OBJ located on the system device. When compiling a program, the Pascal compiler generates subroutine calls to routines in the Pascal support library. The Linker places these routines in the run-time library. The entry points in the library are identified as p\$*nnn* where *nnn* is a small integer. Appendix E of this guide contains a list of these support library entry points. To see these subroutine calls, inspect the MACRO-11 code generated by the Pascal-2 macro switch. Support library routines not called by the compiler have a name instead of a number following the p\$.

Most of the routines in the Pascal support library perform I/O operations or arithmetic computations such as floating-point simulation or trigonometric function approximation. Other routines allocate dynamic memory and report error conditions. Still other routines allow you to change the run-time error reporting to suit your needs. When you build a Pascal job, the Linker searches the Pascal support library for the modules required to run the job. For example, if you compute a logarithm in your program, the Linker includes the support library module that approximates logarithms (\$FLOG, which defines the entry point p\$102).

In most Pascal tasks, the Linker includes from 3K words to 9K words of library modules.

Initialising the Support Library

The Pascal support library is initialized at the start of a Pascal program. When a typical execution begins, the system transfers control to p\$*bgn*, the transfer address of the program, and the support library initialization procedure p\$59 is called. This procedure initializes global variables used by the support library; then it requests that the operating system expand the program code by 4K bytes to make room for the stack, which is originally located in low memory. The routine then moves the stack from low memory to its new location at the end of the program code.

After the stack is repositioned, control transfers to p\$33, the file initialization routine. P\$33 assigns the standard files input and output (channel 16 and 17 for both) to TT:, the terminal. (Channels are discussed below.) The support library then transfers control to the first statement of the program, and execution begins. However, if the program is being debugged with the Debugger, instead of control transferring to the program, control transfers to p\$67, the Debugger initialization routine. This routine initializes the Debugger and opens its files. The Debugger then takes over execution of the program. (See the Debugger Guide for details.)

Support Library Data Definitions

Constants, data and internal file definitions used by the support library are contained in the file LIBDEF.PAS, included in the Pascal-2 distribution kit. In addition to its use with the support library, this file is "included" in the error-reporting module OPERRO.PAS. LIBDEF defines the file variable, the file status block and the library data area.

By using the `%include` directive, users can include LIBDEF.PAS in any program that accesses the support library's work area directly. The example program PCHAN.PAS, below, defines a function named `GetChannel`, which returns the channel number associated with an open file. The program prints the channel numbers for standard files input and output and for two other files opened by the program. Note that the support library stores the channel number times 2 because RSTS requires all

Pascal-2 V2.1/RT-11 Programmer's Guide

channel numbers be doubled. Also noteworthy is the function `loophole`, predefined in the compiler and used in `GetChannel`. See the Language Specification for details on `loophole`.

```
program PrintChannel;
#include 'libdef';
const
  Firstfile = 'FILE1.DAT';
var
  X, Y: text;
  Filename: packed array [1..10] of char;

function GetChannel(var N: text): integer;
var
  F: user_file_variable;  _____ data type defined in LIBDEF.PAS

begin { procedure GetChannel }
  F := loophole(user_file_variable,N);
  GetChannel := F^.channel div 2;  { RSTS uses channel * 2 }
end; { procedure GetChannel }

begin { program PrintChannel }
  writeln('Input is open on channel: ', GetChannel(input));
  writeln('Output is open on channel: ', GetChannel(output));
  rewrite(X,Firstfile);
  writeln('File 1, ', Firstfile,', is open on channel: ', GetChannel(X));
  write('Enter a file name: ');
  readln(Filename);
  reset(Y,Filename);
  writeln('File 2, ',Filename,', is open on channel: ', GetChannel(Y));
end. { program PrintChannel }
```

To compile and execute the program, use this command:

.R PASCAL

*PCHAN

.R LINK

*PCHAN=PCHAN.SY:PASCAL

*~C

.RUN PCHAN

```
Input is open on channel:      16
Output is open on channel:     17
File 1, FILE1.DAT, is open on channel:      15
Enter a file name: FILE2.DAT
File 2, FILE2.DAT, is open on channel:      14
```

NOTE

The definitions in `LIBDEF.PAS` are provided for informational purposes and are subject to change with each release of Pascal-2. Users who desire a more detailed description of the internal workings of the Pascal support library must obtain the library sources.

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

Support Library's Use of Channels

An I/O channel is the means by which RT-11 and the Pascal support library coordinate access to files opened via `reset` or `rewrite`. Channels are numbered 0 through 15 (decimal), which means a Pascal program can have up to 16 files opened at one time. If the program is overlaid, however, RT-11 reserves channel 15 as the overlay load channel, which is then unavailable for use by the program. For overlaid programs, up to 15 files may be open at the same time.

Normally, Pascal users do not need to know which channels are associated with files in their program; all file accessing is handled automatically by the Pascal support library. However, in some special applications, the channel assignments may be important. For example, a user wishing to open a file from Pascal and access the file with specialized MACRO-11 subroutines needs to make sure the same channel is accessed from Pascal and from MACRO-11.

As the preceding example program PCHAN.PAS shows, input is associated with channel 16 and output with channel 17. These channels are not true channels because actual I/O transfers are made through `.TTYIN` and `.TTYOUT` system calls, which do not require channels. They are assigned to input and output so the support library can reference all open files by channel number.

The preceding example program also shows the order in which the Pascal support library allocates channels, from high to low (channel 15 down to channel 0).

If you use `reset` or `rewrite` on the standard files `input` or `output`, or if you allocate another file variable to `TT`: (the console terminal), the support library assigns a channel from its own list of available channels.

NOTE

Because the support library maintains its own list of available channels, it does not use RT-11 system library routines to get the next free channel. Conflicts in channel allocation arise when a Pascal program calls MACRO-11 external modules that open a file on a specific channel and then attempts to open a file on the same channel.

When a file is closed, the channel associated with the file is placed on the list of channels available to the program.

Run-Time Organization

Form of the Generated Code

Pascal-2 code is divided into program sections called "psects." The psects for the main program and any separately compiled procedures are combined with the Pascal support library by the Linker to produce an executable program image. The use of multiple psects arranged in the order the Linker encounters them provides greater flexibility for the combination of individual procedures into a program. (The "blank" psect is normally placed first.)

The compiler generates these psects:

"blank"	The instruction code for the compilation unit. The blank psects for all compilation units are concatenated; compiled code will not attempt to write to this psect.
CONSTS	Contains all constants generated by the compiler. This includes constants declared by constant declarations or implicit in the code. This psect also contains jump tables generated by case statements, so there is a complete separation of instruction references and data references. The CONSTS psects for all compilation units are concatenated; compiled code will not attempt to write to this psect.
DIAGS	Contains line number and procedure name data used in the printing of the run-time walkback. The information is encoded to save space. This psect is not generated if the nowalkback switch is specified in the compilation.
GLOBAL	Contains all global variables used in the main program and external procedures. This psect is arranged so that the global variables are shared among all procedures. The main program and all procedures that reference global variables should have exactly the same declarations. The size of the resulting psect is that of the largest GLOBAL psect generated by any of the compilation units. If the own switch is specified in the compilation, this psect is instead named with the first six characters of the program name, allowing multiple global variable segments. Compiled code will write to this psect.
P\$DYNL	A two-word psect defining a dynamic link to the Post-Mortem Analyzer. (The PMA prints the walkback.) The first word of the psect is a pointer used by the PMA to trace the stack frames for the walkback. With walkback enabled, the second word of this psect contains the address of P\$PMA , the entry point of the PMA. If the nowalkback or nomain compilation switch is used, the second word contains a zero and no jump is made to the PMA.
SHIFTS	Generated only if the target machine does not have the EIS hardware option (sim). This psect contains a table of shift instructions that simulate multiple shifts. The psect is overlaid in a manner similar to TABLES and is treated as read-only by the compiled code.
TABLES	Contains bit tables used for access to set elements and individual bits within a word. All Pascal compilations generate this psect, but all copies will be overlaid by the Linker so that only a single copy will exist in the final program. Compiled code will not attempt to write to this psect.

The following table summarizes the attributes of the various psects. Refer to the MACRO-11 manual for further information on the meaning of the attributes.

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that every entry, no matter how small, should be carefully documented to ensure the integrity of the financial data. This includes recording dates, amounts, and the nature of the transactions.

In the second section, the author outlines the procedures for reconciling the accounts. This involves comparing the internal records with the bank statements to identify any discrepancies. The process should be completed regularly to prevent errors from accumulating.

The third part of the document addresses the issue of budgeting. It suggests that a well-defined budget can help in managing expenses and ensuring that the organization stays within its financial limits. The budget should be reviewed periodically to adjust for any changes in circumstances.

The final section provides a summary of the key points discussed. It reiterates the importance of accuracy, regular reconciliation, and effective budgeting. The author concludes by stating that these practices are essential for the long-term financial health of the organization.

<u>Psect Name</u>	<u>Attributes</u>
"blank"	RW,I,LCL,REL,CON
CONSTS	RW,D,LCL,REL,CON
DIAGS	RW,D,LCL,REL,CON
GLOBAL	RW,D,GBL,REL,OVR
P\$DYNL	RW,D,GBL,REL,OVR
SHIFTS	RW,I,GBL,REL,OVR
TABLES	RW,D,GBL,REL,OVR

So that Pascal programs may be included in libraries, each Pascal-2 object file has a module name consisting of the first six characters of the output file name. Thus a program compiled with the line:

```
.R PASCAL
*RESPROG = HDR,INPROG
```

will have the module name RESPRO.OBJ. This compilation performs "source concatenation." Note that with source concatenation INPROG.PAS must not contain a **program** statement or compilation errors will result.

Memory Organization

On the PDP-11 a program has access to 32768 words (frequently abbreviated to 32K). The exact arrangement of storage is determined by the commands to the Linker, but a typical program may look something like Figure 2-1, which represents a snapshot taken during execution. The numbers are representative; actual values vary from program to program.

RT-11 System Area

The RT-11 System Area occupies the first 256 words of all programs and contains interrupt vectors and status indicators used by RT-11. This area is also used for communication between the Pascal program and other programs linked by chaining.

Program Code

The program code section contains the instructions for the user program. The size of this section is determined by the amount of user code.

Global Variables

The global variables section contains the global variables used by the Pascal main program and external procedures. The size is that of the largest global variable section in any compilation unit.

Constants

The constants section consists of all constants, such as strings or real constants, needed by the program. The section also contains the jump tables for **case** statements. The size of this section is determined by the user code.

Tables

The tables section, which contains data needed by all Pascal programs, is 40 bytes long.

Run-Time Library

This section contains routines from the Pascal run-time library used by a program. Only those routines needed by a particular program are loaded here.

1911

1911

1911

1911

1911

1911

1911

1911

1911

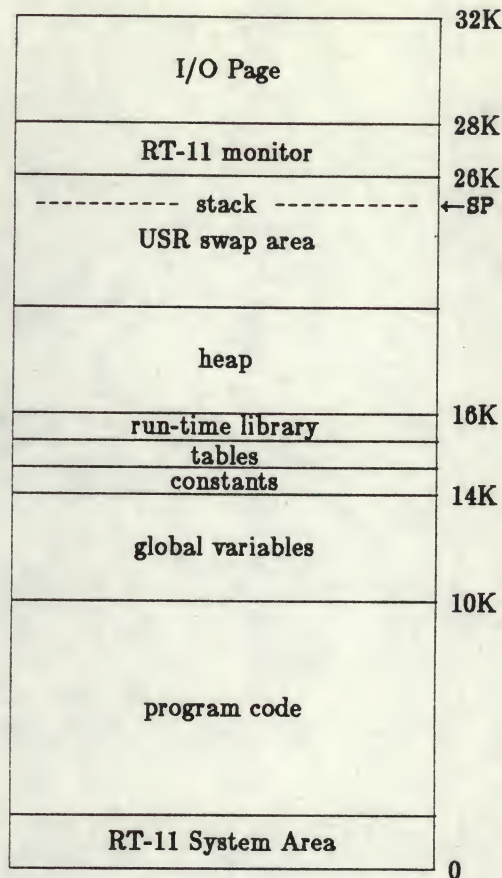


Figure 2-1. Typical Memory Layout of a Pascal Program.

The Stack

The stack contains all variables local to inner blocks of the program, plus parameters, procedure linkage information, and temporary working storage. Upon entry to a procedure or function, space is allocated on the stack (a stack frame) containing space for all storage local to that block. The format of the stack frame is described below.

The stack starts at the highest available address and expands downward, and the heap begins just after the program image and grows upward. This allows the maximum room for growth in both.

The stack pointer (SP) always points to the top of the stack (lowest physical space). If the space available for the stack is too small, the stack pointer will eventually exceed the limits of the stack space and cause the "stack overflow" error.

The Heap

The heap is an area for dynamically allocated memory used for I/O control blocks, buffers, and variables allocated with `new`.

The heap is allocated from the bottom of available memory and can grow until it meets the stack, which is allocated at the top of available memory.

Space is returned to the heap when files are closed or when variables previously allocated with the standard procedure `new` are deallocated with the standard procedure `dispose`. Such space is then

Vol. 40, No. 1, 1965

Published by the American Academy of Arts and Sciences

100 Brookline Avenue, Boston, Massachusetts 02116

Subscription prices: \$10.00 per volume in advance

Single issues: \$2.50 each

Second class postage paid at Boston, Massachusetts

Postmaster: Send address changes in advance

Copyright © 1965 by the American Academy of Arts and Sciences

Printed in the United States of America

The American Academy of Arts and Sciences was founded in 1780 to promote the advancement of knowledge and the improvement of the human condition. It is a non-profit organization, and its members are elected by their peers. The Academy's work is carried out through a variety of programs, including the publication of the *Journal of the American Academy of Arts and Sciences*, the sponsorship of lectures and conferences, and the support of research in the arts and sciences. The Academy's members are drawn from a wide range of disciplines, and its work reflects the interdisciplinary nature of the Academy's mission.

The *Journal of the American Academy of Arts and Sciences* is a peer-reviewed journal that publishes original research and scholarship in the arts and sciences. It is one of the leading journals in the field, and its content is highly influential. The journal's articles are written by leading scholars in the field, and they provide a valuable resource for researchers and students alike. The journal's focus is on the advancement of knowledge and the improvement of the human condition, and its content reflects this commitment.

available for further heap allocation. The error message "not enough memory" results if no space is available to satisfy a request for heap storage.

If a program running under XM is linked as a virtual overlay job, the overlays can be structured in such a way that the excess (and unused) space in the null root is added to the heap's "free list" of available memory. See "The Linker, Overlays, and the Librarian" for details.

For information on ways to monitor the allocation of the stack and heap at run-time, refer to the section on "Monitoring Memory Usage" later in this guide.

The Monitor and I/O Page

For RT-11 systems other than XM virtual jobs, the monitor space contains the RT-11 resident monitor, the user service routine (USR) if it is set "NOSWAP," and any device handlers loaded with the LOAD command. The I/O page contains device status and command registers.

For an RT-11 XM virtual job (bit 10 set in the JSW), the monitor and I/O page are not allocated, and the stack will begin at the top of user memory. See "Virtual Jobs and the XM Monitor" for details.

The Stack Frame

As each procedure or function is entered, space is allocated on the stack for parameters, linkage data, and local use. This space is called a "stack frame"; the "stack" consists of these stack frames.

Figure 2-2 shows the format of a stack frame.

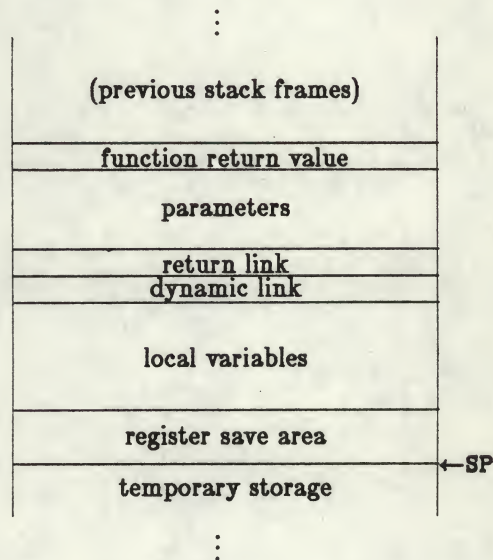


Figure 2-2. Format of a Stack Frame.

Not all of the fields will be used by the compiler for every procedure; only the return link is present in every frame. It is the responsibility of the called procedure to remove the parameters and local variables from the stack before a return is made to the caller.

1. The first part of the report deals with the general situation of the country and the progress of the work.

2. The second part of the report deals with the results of the work and the progress of the work.

3. The third part of the report deals with the results of the work and the progress of the work.

4. The fourth part of the report deals with the results of the work and the progress of the work.

5. The fifth part of the report deals with the results of the work and the progress of the work.

6. The sixth part of the report deals with the results of the work and the progress of the work.

7. The seventh part of the report deals with the results of the work and the progress of the work.

8. The eighth part of the report deals with the results of the work and the progress of the work.

9. The ninth part of the report deals with the results of the work and the progress of the work.

Function Return Value

The function return value field appears only for functions. A value assigned to the function name within the function will be stored in this location and left on the top of the stack when the function returns. Space for this field is allocated by the calling routine before evaluation of the arguments for the function call. The space is removed from the stack when the calling routine has no further use for the value.

Parameters

The parameter area has an entry for each parameter to the procedure or function. The entry for a value parameter will contain the value of the corresponding argument, while the entry for a variable parameter will contain the address of the argument. Parameters are pushed onto the stack as they are evaluated, in left to right order, so the first parameter to a procedure will be the first one pushed onto the stack.

Return Link

The return link is the address to which control will be transferred on return from the procedure or function.

Dynamic Link

By default, procedures compiled with **walkback** enabled establish a dynamic link that points to the dynamic link in the previous stack frame. The base of the linked list of stack frames is contained in the first word of the psect **P\$DYNL**. When a run-time error is detected, the dynamic link is used to show the procedure calls that led to the error (the procedure walkback). A dynamic link is not present in the stack frame for procedures compiled with **nowalkback**.

Local Variables

The local variable field contains space for all local variables of the procedure or function. The field is allocated upon entry to the block.

Register Save Area

This area saves the values of all registers used within the procedure. The registers are saved on entry to the procedure and restored on exit. Only registers actually used are saved. The general registers are stored first, with the highest register used pushed first. (This is important to the algorithm for locating variables in lexically enclosing blocks.)

Temporary Storage

In the process of generating code, expressions that are used more than once are computed and the values saved. These values may be saved on the stack if no register is available to hold them. Also, the stack is used to interface with support library routines and the operating system.

Handwritten text at the top of the page, possibly a header or title.

Handwritten text in the upper middle section of the page.

Handwritten text in the middle section of the page.

Handwritten text in the lower middle section of the page.

Handwritten text in the lower section of the page.

Handwritten text at the bottom of the page.

Handwritten text at the very bottom of the page.

Monitoring Memory Usage

An executable Pascal program is typically arranged in memory with the program code written to low memory, followed by the memory area shared by the stack and heap. The I/O page and monitor are loaded into the high end of memory. (In this discussion, a typical Pascal program uses no Linker options to rearrange memory.) The remaining memory is unallocated and available for the stack and heap.

To arrive at this arrangement, the following events occur:

1. The program code is loaded into memory, with the stack in low memory.
2. The support library initialization procedure is called, initializing the support library and performing the next three steps. (See the "Support Library" section.)
3. The stack is moved to its new location at the highest available memory location with the `.SETTOP -2` directive or at the address specified on the `/M:n` Linker option.
4. `P$KORE`, the pointer to the top of the heap, is initialized. The heap immediately follows the program code.
5. The user program is started. During execution, if the stack and heap meet, the program terminates with either of two errors, "not enough memory" or "stack overflow," depending on the cause.

Although overlaid programs differ from non-overlaid programs in their arrangement in memory, the same sequence of events occur to load them into memory. In this case, the program code is made up of a root segment and a number of overlay regions into which the program is divided at link-time. For low memory overlays, if the heap is exhausted, no more memory is available and the program terminates. For extended memory overlays (virtual overlays), the excess memory in the root segment is placed on the heap's free list, extending the memory available for the heap. This and other information on overlays is described in "The Linker, Overlays, and the Librarian" and later in this section.

In certain applications you may find it advantageous to keep track of the stack and heap as they are used by a program. The Pascal support library contains three routines — `space`, `p$inew` and `p$dispose` — that allow you to keep track of memory allocated to the stack and heap. Briefly, the `space` function returns the amount of stack and heap space available to an executing program at a particular time. The `p$inew` function returns the address of a block of memory having a specified length. The `p$dispose` procedure deallocates blocks of memory allocated by `p$inew`. In a later example a boolean function `NewOK` is provided, which not only shows the correct way to use the three routines but also is useful in determining whether enough memory is available to satisfy a request for a block of memory.

Two reasons for monitoring the size of the stack and heap are to find out how close the program is to running out of memory, and to find out whether enough memory is available to perform a given subtask. For example, a checkers-playing program could use these functions (as in `NewOK`) to determine the number of moves that the program can look ahead based on the amount of memory available to perform the look-ahead.

`Space` can be called independent of the other two support library routines, whereas `p$dispose` must be used to deallocate memory allocated by `p$inew`. The routines are described in detail below.

[Faint handwritten text at the top of the page, possibly a title or header.]

[First paragraph of faint handwritten text.]

[Second paragraph of faint handwritten text.]

[Third paragraph of faint handwritten text.]

[Fourth paragraph of faint handwritten text.]

[Fifth paragraph of faint handwritten text.]

[Sixth paragraph of faint handwritten text.]

[Seventh paragraph of faint handwritten text.]

[Eighth paragraph of faint handwritten text.]

[Ninth paragraph of faint handwritten text.]

[Tenth paragraph of faint handwritten text.]

The 'Space' Function

The **space** function is used to determine the amount of stack and heap space available to an executing program. With this function you can determine how close the program is to running out of memory.

The function **space** must be declared **external** to the program, as shown:

```
function Space: functype; external;
```

where *functype* is the data type of the function and its returned value. The function is usually declared of type **integer** but another data type similar to **integer**, "unsigned" (0..65535), can be used to represent the value. The value returned by **space** depends on the amount of space allocated to the stack and heap.

Figure 2-3 is a memory diagram showing a program's arrangement in memory and its allocation of stack and heap space. The diagram also shows the relationship between the stack and heap and the returned value of the **space** function. This diagram is designed as an aid in visualizing the use of the stack and heap and as an aid in understanding the way the value of **space** is arrived at.

As is the case on RT-11, the stack and heap share the same block of memory. The stack begins at the high end of the stack/heap area and grows down; the heap begins at **P\$KORE**, the low end of the block and grows up. The **space** function monitors the use of the stack and heap, returning the difference between the top of the stack (**SP**) and the top of the heap (**P\$KORE**).

The figure is divided into "times," or snapshots of a program in memory at various stages of its execution. Time 0 is the point at which the user program actually begins execution, after the program code is loaded into memory and the support library is initialized.

The figure contains the following five symbols. Curled braces designate the value that the **space** function would return at the indicated time. Vertical arrows represent stack and heap expansion. **SP** signifies the stack pointer. **P\$KORE** is a pointer to the top of the heap. In general, **KB** stands for K bytes; **64KB** stands for 64K bytes or 32K words, the high end of memory; **56KB** stands for 56K bytes or 28K words, the end of the RT-11 monitor and beginning of the I/O page; **52KB** stands for 52K bytes or 26K words, the bottom of the stack and the beginning of the monitor; **0KB** denotes the beginning of memory (low core and vectors). These memory locations are typical of Pascal programs but may vary from program to program.

Be aware of the fact that the **space** function does not account for the fragmentation of the heap as a result of calls to **new** and **dispose** and the opening and closing of files. Unused portions between the low and high end of the heap are treated as if they are allocated. Again, for extended memory overlays the **space** function does not take into account the memory added to the heap's free list.

See the example under "Example: Function NewOK" for use of **space**.

Function 'P\$inew' and Procedure 'P\$dispose'

The function **p\$inew** and procedure **p\$dispose** are entry points for the standard procedures **new** and **dispose**. **P\$inew** allocates a specific sized block of memory, and **p\$dispose** deallocates a specific block. As a comparison, the standard procedures **new** and **dispose** determine the size of the block to allocate or deallocate from the length of the data type.

An example use of **p\$inew** and **p\$dispose** is a cache system in which a program needs to use as much memory as is available for data storage before it writes the data to a file.

Handwritten text, likely a letter or document, written in cursive script. The text is extremely faint and illegible due to fading or poor reproduction quality. It appears to be a single page of writing, possibly a letter, given the presence of two binder holes on the right side.

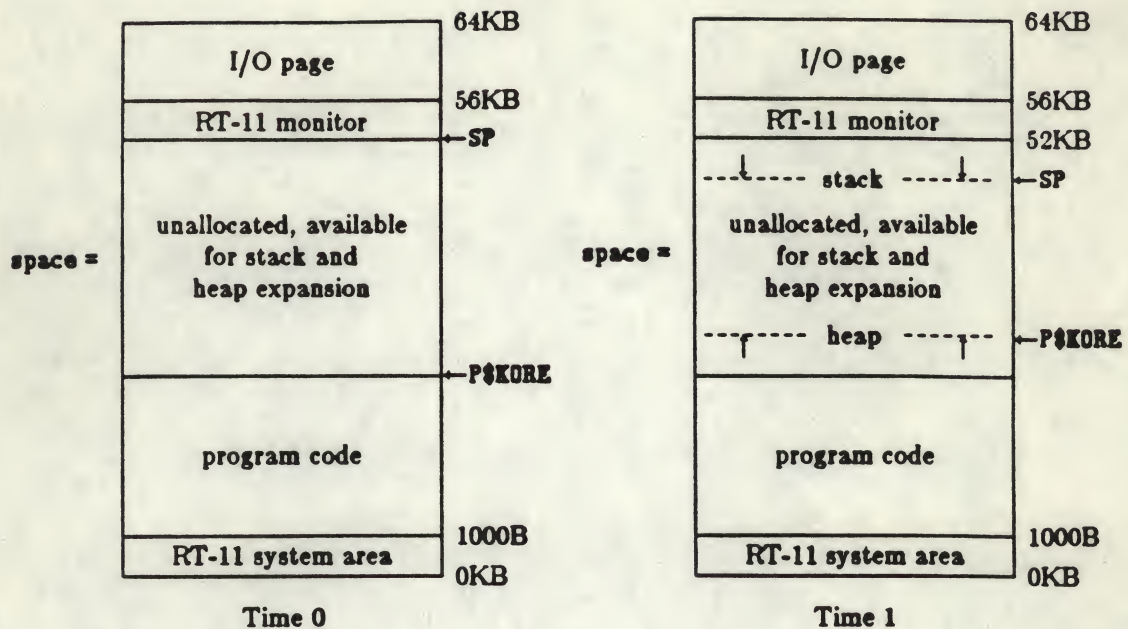


Figure 2-3. Tracking Memory Usage with the SPACE function.

At Time 0, the support library has been initialized but no Pascal statements have been executed. The stack and heap, in sharing the same block of memory, grow toward each other and ideally never meet. If `space` is called at Time 0, the value it returns is equal to the size of the stack and heap area (SP minus P\$KORE). At Time 1, the amount of stack and heap remaining and the value of `space`, being identical, both decrease by the same amount as stack and heap space is allocated. As time progresses, memory available for the stack and heap continues to be used and disposed of until the program ends or until the allotted memory is exhausted and the program aborts.

Defined in the support library, these two routines must be declared **external** to the program, as shown:

```
function P$inew(blocksize: argtype): functype; external;

procedure P$dispose(pointer, blocksize: argtype); external;
```

where

blocksize is the size of the memory block to be allocated or deallocated, in bytes.

pointer is the address of block to deallocate.

argtype is the data type describing *size* and *pointer*.

functype is the data type of the function's return value. The returned value is the address of the block. If there is not enough contiguous memory available to satisfy the request, a value of 0 is returned for numeric data types, nil for pointer types. The most common types used with `p$inew` and `p$dispose` are the standard type **integer** and a user-defined type **unsigned**, in the range 0..65535.

For an example showing the use of `p$inew` and `p$dispose` see the code for procedure `New0K`, below.

Jan 1	100	100	100
Jan 2	100	100	100
Jan 3	100	100	100
Jan 4	100	100	100
Jan 5	100	100	100
Jan 6	100	100	100
Jan 7	100	100	100
Jan 8	100	100	100
Jan 9	100	100	100
Jan 10	100	100	100
Jan 11	100	100	100
Jan 12	100	100	100
Jan 13	100	100	100
Jan 14	100	100	100
Jan 15	100	100	100
Jan 16	100	100	100
Jan 17	100	100	100
Jan 18	100	100	100
Jan 19	100	100	100
Jan 20	100	100	100
Jan 21	100	100	100
Jan 22	100	100	100
Jan 23	100	100	100
Jan 24	100	100	100
Jan 25	100	100	100
Jan 26	100	100	100
Jan 27	100	100	100
Jan 28	100	100	100
Jan 29	100	100	100
Jan 30	100	100	100
Jan 31	100	100	100

The following table shows the results of the experiments conducted during the month of January. The first column gives the date, the second column the number of trials, the third column the number of successes, and the fourth column the percentage of successes. The results show that the success rate was generally high, with a slight decrease towards the end of the month.

The following table shows the results of the experiments conducted during the month of February. The first column gives the date, the second column the number of trials, the third column the number of successes, and the fourth column the percentage of successes. The results show that the success rate was generally high, with a slight decrease towards the end of the month.

The following table shows the results of the experiments conducted during the month of March. The first column gives the date, the second column the number of trials, the third column the number of successes, and the fourth column the percentage of successes. The results show that the success rate was generally high, with a slight decrease towards the end of the month.

The following table shows the results of the experiments conducted during the month of April. The first column gives the date, the second column the number of trials, the third column the number of successes, and the fourth column the percentage of successes. The results show that the success rate was generally high, with a slight decrease towards the end of the month.

Example: Function 'NewOK'

The boolean function **NewOK**, provided below as an external, uses the three routines previously described functions to determine whether a block of memory can be allocated, leaving a specified amount of stack space. We recommend that you reserve 200₈ to 1000₈ bytes of stack space for parameter and local variable storage and for error processing. The amount of memory you reserve depends on the parameter and local variable requirements of the procedures being called by the program. For instance, if the program calls numerous procedures, each containing a large number of parameters and local variables, the amount of memory to reserve would be greater than for a program that uses smaller procedures.

NOTE

If a program that uses **NewOK** aborts with a "stack overflow" error, the amount of reserved memory is probably not large enough for the amount of stack space required for parameters and local variables. To alleviate this error, increase the amount of memory you are reserving for the stack.

In addition to showing the correct way to use the three routines, **NewOK** can be incorporated into your programs when you need to determine if a request for memory will fail.

NewOK, which could be stored in **NEWOK.PAS**, returns a **true** value if the block can be allocated, **false** if the block cannot be allocated. The first argument to **NewOK** is the size, in bytes, of the memory to be allocated. Use the **size** function to determine the size of a Pascal data type. (The **size** function is described in the Language Specification.) The second argument is the amount of stack space in bytes that you want to remain unallocated, if the block is allocated.

First, the function allocates the desired block of memory with the call to **p\$inew**. If **p\$inew** returns a zero, this means that there was not enough memory available to allocate a block of that size, and **NewOK** returns **false**. However, just because the block was allocated does not necessarily mean that **NewOK** returns **true**. If the returned value of **space** is less than the amount of stack space you have reserved for variables, etc., **NewOK** is set to **false** because memory would be taken from the reserved block. Of course, the function returns a **true** value if the block was safely available. Finally, the same block of memory is disposed of by **p\$dispose**. (Remember, **NewOK** only checks to see if a block of memory could be allocated. To allocate the block, call **new**.)

```
{ $nomain }
type
  Unsigned = 0..65535; { Unsigned integer }

function p$inew(Blocksize: Unsigned): Unsigned;
  external;          { Allocate a specific sized block of memory }

procedure p$dispose(Pointer, Blocksize: Unsigned);
  external;          { Dispose of a specific sized block of memory }

function space: Unsigned;
  external;          { Determine amount of stack space left }
```

CHICAGO, ILLINOIS

Dear Sir: I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the matter of the purchase of the land for the proposed University of Chicago. I am sorry to hear that you are not satisfied with the price offered for the land. I am sure that the University will be glad to pay a reasonable price for the land, and I am sure that you will be glad to sell it at a reasonable price. I am sure that we can reach an agreement on this matter.

I am, Sir, very respectfully,
Yours very truly,
The University of Chicago

I am, Sir, very respectfully,
Yours very truly,
The University of Chicago

Very truly yours,

THE UNIVERSITY OF CHICAGO

CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO


```

function Newok(Reserved, Stackspace: Unsigned): boolean; external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

function NewOk;

var
  P: Unsigned; { Address of block if allocated }

begin { NewOK }
  P := p$inew(Reserved);           { Try to allocate block }
  if P = 0 then NewOK := false      { No luck }
  else
    begin
      NewOK := space >= Stackspace; { Check for enough stack left }
      p$dispose(P, Reserved);        { Deallocate the block }
    end;
end; { NewOK }

```

To show the correct way to set up the critical variables and call `NewOK`, a checkers-playing program could use `NewOK` to find out how many moves it can look ahead. The sample program provided below does just that. For illustrative purposes, the program, `CHECKT.PAS`, simulates a real checkers program, making use of a 10,000-word array `Dum` to produce a larger task size. `CHECKT` uses a linked list to store the look-ahead moves a normal checkers program would make. The program merely illustrates the use of `NewOK` to perform the look-ahead.

```

program CheckTest;

const
  Reserved = 200; { amount of stack space to reserve }

type
  Unsigned = 0..65535;
  Ptr = ^Node;           { Pointers into search tree }
  Node = record           { Node in search tree }
    Father: Ptr;          { Father of this node }
    Son: Ptr;             { Pointer to best son }
    Brother: Ptr;         { Link to next brother }
    Value: integer;       { Value of this board position }
    Move: integer;        { Move descriptor to reach node }
    Jump1, Jump2: integer; { Jumped pieces removed by move }
    Mobility: integer;    { Mobil and deny }
    Attack: integer;      { Pin and threat }
    Gradient: integer;    { Target gradient }
    Bits: integer;        { Scoring bits }
  end;

var
  Alloc: boolean;
  P: Ptr;
  Movesize: unsigned;
  NumMoves: unsigned;     { number of moves }
  Nextmove: node;
  Dum: array [1..10000] of integer; { Dummy array simulating a long program }

```

Handwritten text at the top of the page, possibly a title or header.

Handwritten text in the upper middle section.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text at the bottom of the page, possibly a footer or concluding remarks.

Pascal-2 V2.1/RT-11 Programmer's Guide

```
function NewOK(Reserved, Stackspace: Unsigned): boolean;
  external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

begin { CheckTest }
  Movesize := size(Node);
  writeln('The size of a move is: ', Movesize:1);
  new(P);
  NumMoves := 1;
  repeat
    alloc := Newok(Movesize, Reserved);
    if alloc then begin
      new(P^.son);
      P := P^.son;
      NumMoves := NumMoves + 1
    end;
  until not alloc;
  write('The number of moves necessary to fill up the heap is: ');
  writeln(NumMoves:1);
end. { CheckTest }
```

The compilation process for this program is as follows:

.R PASCAL

*NEWOK

.R PASCAL

*CHECKT

.LINK CHECKT=CHECKT, NEWOK, SY:PASCAL

.RUN CHECKT

The size of a move is: 22

The number of moves necessary to fill up the heap is: 1141

Handwritten text at the top of the page, possibly a date or header.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Fifth main paragraph of handwritten text.

Sixth main paragraph of handwritten text.

Seventh main paragraph of handwritten text.

Eighth main paragraph of handwritten text.

Storage Allocation

The compiler assigns storage for variables of pre-declared types as shown in this table:

<u>Type</u>	<u>Size (bytes)</u>	<u>Alignment (bytes)</u>
Boolean	1	1
Char	1	1
Integer	2	2
Real	4	2 (\$double off)
Real	8	2 (\$double on)
Text	2	2

Space for user-defined types is allocated as follows:

Enumeration

If the type has up to 256 members, it is allocated one byte aligned on a byte boundary. If it has more than 256 members, it is allocated two bytes, aligned on a two-byte boundary.

Subrange

Allocated in the same way as the parent type.

Pointer Allocated two bytes, aligned on a two-byte boundary.

Array Allocated the amount of space needed to hold the number of elements specified, aligned in the same way as the element type. The elements are placed in ascending memory locations.

Set Allocated one bit for each member of the base type, with the total size rounded up to the next larger full byte. Bit allocation begins with the least significant bit of the first byte. If the size is a single byte, it is aligned on a byte boundary; otherwise it is aligned on a two-byte boundary. A base type that is a subrange is expanded to the full range of possible values before the set is allocated. For example:

```

type
  Color = (Red, Orange, Yellow, Green, Blue);
  Hot = Red..Yellow;

  Colorset = set of Color;
  Hotset = set of Hot;

```

In this example, **Hotset** is allocated the same amount of space as **Colorset**. A maximum of 256 members is allowed; a base type of integer, or any integer subrange, has members from 0 to 255.

Record Each field in the record is allocated space in the same way as a variable of the same type, in the order specified. The alignment of the record is the maximum of the alignments of its fields.

Packed Array

The number of bits needed to contain each element is computed. For example, the subrange type 0..3 requires two bits to contain a value. If the space required for an element is less than a word, the element size is increased to the smallest power of two bits (1, 2, 4, 8, 16) that will contain the value. The array is allocated space to hold the number of elements specified, where each element is considered to be of the size just computed. If elements are allocated eight bits or less, the array is aligned on a byte boundary. If the elements require a word or more, space is allocated as for a normal array type.

Packed Set

The same as unpacked sets, except that the size is not rounded up to an even byte and the alignment is to a byte boundary.

Packed Record

Each field in the record is allocated exactly the number of bits required to contain it, except that a field of a simple type that would span or cross a word boundary will be forced to begin at a word boundary. Fields are allocated in the order declared, beginning at bit zero (least significant bit).

NOTE

The predefined functions **size** and **bitsize** will report the amount of storage allocated to any user-defined structured type. See the Language Specification for details.

Run-Time Error Reporting

The Pascal-2 run-time error reporting system is intended to simplify error analysis by reporting run-time errors in terms of source lines and procedure names. Upon detecting a run-time error, the reporting system prints a short description of the error, then traces the execution history, procedure by procedure, from the point of error back to the main program. This is called a "walkback," or "traceback."

Errors are detected by the hardware or by special checks inserted in the generated code. After an error is detected, control of the program then passes to an error routine, which closes all open files and prints an error message and stack traceback.

The walkback consists of the following:

- The message header, which includes the program name, type of error, and program counter at the time of the error. The two types of run-time errors are "fatal" and "I/O." Fatal errors are unrecoverable; I/O errors are recoverable. For details on I/O-error recovery, see "I/O Error Trapping" later in this section. The program counter is the location at which the error occurred. If you utilize the walkback, the program counter is of little use to you since the location of the error is given as a line number in a procedure.
- A description of the error. See Appendix B of this guide for a detailed explanation of the error messages. By modifying OPERRO.PAS, you can change the wording of Pascal run-time messages if you so desire. See the section on "Customizing Error Reporting" for details.
- For I/O errors, the error code and the file name of the file causing the error. The error code is printed in both decimal and octal. On RT-11, all I/O error codes are errors detected by the support library and are described in Appendix B of this guide. The file name includes such information as the device name, file name and extension.
- The location of the error in terms of line number and procedure name. The line number refers to lines in the overall source program, not statements in individual procedures. (For external procedures, the line number refers to lines in the external module.) A special case arises when a run-time error is detected in an external procedure compiled with **nowalkback**. In this case, the location of the error is given as an octal address.
- The reverse sequence of active procedure calls back to the main program, if the error occurred at a level other than the main program.

The walkback can be disabled at compile time; to do this, use the **nowalkback** compilation switch. When **nowalkback** is selected, the message header and the error message are printed but not the walkback.

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

Pascal-2 V2.1/RT-11 Programmer's Guide

Examples

Example 1.

This example provides a look at a possible run-time error condition and the resulting error walkback.

.RUN PACKER

PASCAL--Fatal error at user PC= 2316B
Array subscript out of bounds

Error occurred at line 140 in procedure arrangetree
Last called from line 326 in procedure switchnodes
Last called from line 402 in procedure getdep
Last called from line 579 in program packer

Example 2.

A procedure called recursively may have many consecutive activations. In this case, the number of identical lines is indicated by the note (nn times) after the location description. Appearing below is the walkback of a program that looped recursively until the stack overflowed.

.RUN EXTRA

PASCAL--Fatal error at user PC= 5223B
Stack overflow

Error occurred at line 124 in procedure walk
Last called from line 290 in procedure reanalysis (898 times)
Last called from line 423 in procedure unloadbits
Last called from line 440 in procedure matrixmask
Last called from line 535 in procedure processleftop
Last called from line 608 in program extra

Example 3.

This example illustrates the walkback produced as a result of an I/O error that is not trapped by the user.

.RUN REFORM

File to reformat: LSAT.TXT

PASCAL--I/O error at user PC= 2166B
Can't open file
I/O error code= 11. (13B) in file: DK:LSAT.TXT

Error occurred at line 44 in procedure openfile
Last called from line 69 in program reformatter

Handwritten header text at the top of the page.

Handwritten text, possibly a date or reference number.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Fifth main paragraph of handwritten text.

Sixth main paragraph of handwritten text.

Seventh main paragraph of handwritten text.

Eighth main paragraph of handwritten text.

Ninth main paragraph of handwritten text.

Tenth main paragraph of handwritten text.

Eleventh main paragraph of handwritten text.

Example 4.

This example shows the walkback produced as a result of a run-time error in an external procedure compiled with `nowalkback`. Note that if the external procedure had been compiled with `walkback` (the default), the location of the error would be in source terms.

.RUN DIFFS

PASCAL--Fatal error at user PC= 1336B
Division by zero

Error occurred at location 1336
Last called from line 32 in program diffs

I/O Error Trapping

Pascal-2 permits you to write programs that trap and detect many kinds of I/O-related errors that normally would be fatal. Three predefined routines — procedure `noioerror` and functions `ioerror` and `iostatus` — facilitate this trapping of I/O errors. Using these routines, you have the ability to process I/O errors with your own code. You have two options: terminate the program at the occurrence of an I/O error (you can print your own diagnostics), or continue execution in spite of the error. The choice depends on the need.

Since these three routines are predefined in the compiler, they do not need to be declared in your program. They accept a file variable as their only parameter. Details are supplied below.

procedure `noioerror`:

Specifies that the calling program will handle any I/O errors that result from reading or writing to the specified file. The file must be open before `noioerror` is called. This procedure performs the same function as the `/go` file control switch on `reset` and `rewrite` statements.

function `ioerror`:

Determines the status of the last I/O operation that the program performed on the specified file. This `boolean` function returns a `true` value if an I/O error has occurred, `false` if the operation was successful.

function `iostatus`:

Returns the integer error code that describes the last attempt to access a file. This function helps your program determine the cause of the error. Your program can either bypass the problem and continue processing, or terminate so you can correct the problem. The I/O error is detected by the Pascal-2 support library. Pascal-2 error codes, along with the text of the error message and a brief explanation of the cause, are listed in Appendix B of this guide.

When you call these routines, you are responsible for checking the status of each I/O operation, to ensure that it was successful. If you fail to check the status and an error occurred, the results are unpredictable.

The following program illustrates the use of these procedures. The program is designed to continue executing despite an I/O error. The call to `noioerror` indicates to the run-time system that errors

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the above matter.

I am sorry to hear that you are not satisfied with the results of the examination. I have been very anxious to see that the work was done to the best of my ability.

I have been very busy lately, and have not had time to devote to this matter as much as I would like. I am sure that the results will be satisfactory in the future.

I am very sorry to hear that you are not satisfied with the results of the examination. I have been very anxious to see that the work was done to the best of my ability.

I have been very busy lately, and have not had time to devote to this matter as much as I would like. I am sure that the results will be satisfactory in the future.

I am very sorry to hear that you are not satisfied with the results of the examination. I have been very anxious to see that the work was done to the best of my ability.

I have been very busy lately, and have not had time to devote to this matter as much as I would like. I am sure that the results will be satisfactory in the future.

I am very sorry to hear that you are not satisfied with the results of the examination. I have been very anxious to see that the work was done to the best of my ability.

I have been very busy lately, and have not had time to devote to this matter as much as I would like. I am sure that the results will be satisfactory in the future.

I am very sorry to hear that you are not satisfied with the results of the examination. I have been very anxious to see that the work was done to the best of my ability.

I have been very busy lately, and have not had time to devote to this matter as much as I would like. I am sure that the results will be satisfactory in the future.

detected on the standard file input will be handled by the program.

```

program Iotest;

var
    I, Times: integer;

begin
    Noioerror(input);
    for Times := 1 to 4 do
        begin
            write('Type an integer: ');
            read(I);
            if Ioerror(input)
            then writeln('Error detected. Status=', Iostatus(input))
            else writeln('The integer was: ', I: 1);
            readln;
            writeln;
        end;
    end.

```

If this program is compiled and run, the following results might be produced. The first entry results in a successful read of the integer I. The second and third entries result in a Pascal-2 run-time error. See Appendix B for a list of run-time error messages and associated numbers. The final entry is successfully read, and the program ends. (Under normal conditions, the first error would cause the program to abort.)

```

.RUN IOTEST
Type an integer: 1234
The integer was: 1234

Type an integer: 123456789
Error detected. Status=      23

Type an integer: FFG
Error detected. Status=      23

Type an integer: 77
The integer was: 77

```

The I/O error-trapping procedures can be used to determine the reason that a file could not be opened. To use this feature, specify the fourth parameter on calls to **reset** and **rewrite**. Specifying this fourth parameter keeps the **reset** or **rewrite** from trapping a normally fatal "open" error. This allows your program to recover and continue, or terminate under your control.

The following program illustrates the use of **ioerror** and **reset/rewrite**. The program attempts to open a file called TEST.DAT on the device **XXXX:**, a fictitious device name. The error is detected

by Ioerror.

```

program Opnerr;

var
  F: text;
  Status: integer;

begin
  reset(F, 'XXXX:', 'test.dat', Status);
  if Ioerror(F) then
    writeln('I/O status=', Iostatus(F));
end.

```

When this program is compiled and executed on RT-11, it yields the following output. The value 11 is the support library run-time error code for "can't open file."

```

.RUN OPNERR
I/O status=      11

```

Customizing Error Reporting

The flexibility of the Pascal-2 run-time organization allows you to not only handle I/O errors in your code but to customize the run-time error diagnostics to suit your needs. Included in the distribution kit are two Pascal source files, OPERRO.PAS and UERROR.PAS, which let you modify the way in which errors are reported. The object-file equivalents of these two procedures are in the Pascal support library. The changes you make can be used for a one-time debugging run, or they can be permanently installed in the support library.

OPERRO.PAS contains the entry point **P\$ERROR**, which the support library's error-handling routine calls to print the message header and text of the run-time error message. This source file is provided so you can change the wording of any error message simply by editing the source.

UERROR.PAS contains the entry point **P\$UERROR**, which is called following **P\$ERROR** to print additional information about the error. This procedure contains two boolean constants set to **false** in the release version, each controlling (inhibiting) the printing of a separate set of diagnostics. Initially, with the booleans set to **false** nothing is printed. But by editing UERROR.PAS and setting one or more constants to **true**, you can receive a file dump of the offending file and/or a memory map of the program. The **const** fragment below shows the two constants.

```

const
  Dump_Memory = false; { Print a memory map }
  Dump_File = false; { Print detailed file dump }

```

By using UERROR.PAS, you can add your own code to UERROR.PAS for the printing of more specialized debugging information, and you can print out the values of critical variables used in the program. To print variables you must include the program's global variable declarations in UERROR.PAS. The ability to print critical variables is useful when you have a program with many overlays or when the program is too large to run with the Debugger.

When a run-time error is detected, several steps are taken:

1. Control of the program transfers to the **\$ERR** error-control module, in the Pascal support library. **\$ERR** collects information about the error from the library data area.
2. **\$ERR** calls **P\$ERROR** (in OPERRO.PAS) to print the message header followed by the error message.

Pascal-2 V2.1/RT-11 Programmer's Guide

3. \$ERR then calls P\$UERROR (in UERROR.PAS), which does nothing (by default) but can be modified to print a memory map of your program and/or dump the contents of the offending file.
4. On return from P\$UERROR, the error-control routine \$ERR transfers control to the Post-Mortem Analyzer (PMA), which prints the error walkback, and the program terminates.

When you use a modified version of one or both of these procedures as externals, you need not declare them explicitly in the main program. After the altered version(s) of these procedures are compiled (with `nomain`), simply specify the module name(s) on the Linker command line after the program name. The Linker substitutes your version for the version in the support library. This sequence of commands should be used:

```
.R PASCAL  
*PROG  
.R PASCAL  
*UERROR  
.R PASCAL  
*OPERRO  
  
.R LINK  
*PROG=PROG,UERROR,OPERRO,SY:PASCAL
```

Another way to override the version in the support library is to include the modified OPERRO.PAS and/or UERROR.PAS as part of the main program. The `%include` directive does this easily. (See "Implementation Notes" in this guide for use of `%include`.)

For example:

```
%include 'operro';  
%include 'uerror';
```

When using the `%include` directive, compile and link the main program as you normally would. The Linker resolves the references to P\$ERROR and P\$UERROR with the procedures included in the program. The program PROG, above, would be linked in this manner with these commands:

```
.R PASCAL  
*PROG  
  
.R LINK  
*PROG=PROG,SY:PASCAL
```

The constant `Dump_Memory`, if set to `true`, causes the program to print a memory dump, or map, of the program showing the program's use of memory. The map is printed by the external procedure `memmap`, a Pascal support library routine. Although `memmap` is declared in UERROR.PAS, you can use it with any Pascal program, independent of UERROR.PAS. Simply declare it in the program as an external with no parameters. The map helps you determine the way dynamic memory is being

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

Run-Time Error Reporting

allocated and perhaps the reason your program is running out of memory.

.RUN DIAL

PASCAL--Fatal error at user PC= 1276B
Attempted reference through NIL pointer

Memory Map:

Pointer to top of stack - RESR6 : 140060B
Pointer to global data - RESR5 : 13102B
Standard Output :13076B
Standard Input :13100B
Saved Output~ :33256B
Saved Input~ :33316B
Last file access :
 File @ 33256B TI : .
Active files :
 File @ 33316B TI : .
 File @ 33256B TI : .
PMA active
No free list
No orphan list
Last User PC 5116B
Pointer to top of heap (P\$KORE) : 33556B

Error occurred at line 29 in program dialphones

The constant `Dump_File` can be used to print a detailed dump of the Pascal and RT-11 file structures when an I/O error is detected. When `Dump_File` is set to `true`, the support library module `fdump` is called to dump information about the file. Programmers familiar with the structure of the file variable may find this information useful in diagnosing obscure file problems.

The following listing is the file dump produced by `fdump`. Although you wouldn't want the file dump printed at each occurrence of a run-time I/O error, in certain circumstances the file dump may help you diagnose more obscure errors. Lines such as "buffer size" display the information first in octal,

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that proper record-keeping is essential for the transparency and accountability of the organization. The text further outlines the specific procedures for recording and verifying financial data, ensuring that all entries are supported by appropriate documentation.

In the second section, the document addresses the challenges associated with managing a large volume of data. It highlights the need for efficient data management systems and the importance of regular data audits to identify and correct any discrepancies. The text also discusses the role of technology in streamlining data collection and analysis processes.

The third part of the document focuses on the importance of communication and collaboration among all stakeholders. It stresses that clear and consistent communication is vital for ensuring that everyone is on the same page and working towards common goals. The text provides guidelines for effective communication, including the use of clear language and active listening.

Finally, the document concludes by reiterating the commitment to high standards of integrity and ethical conduct. It states that all actions must be guided by the organization's core values and principles, and that any violations will be dealt with strictly according to the established policies.

Pascal-2 V2.1/RT-11 Programmer's Guide

denoted by the B, then in decimal.

.RUN COVER

PASCAL--I/O error at user PC= 1072B
File is not a random access file. Use /SEEK
I/O error code= 27. (33B) in file: DK:LSAT.DAT

File information for file variable at:34144B
Contents of file variable:
Ptr: 34204B Pointer to data in file buffer
Stat:250B File status

Records are Blocked
Sequential File
Read operations permitted
Pending End of File
Permanent File
Non interactive device/.read
Text file
Current character not defined

Name: DK :LSAT .DAT
File Size in Blocks: 1B 1.
Channel : 15
Current Block : 0B 0.
Buffer Address : 34204B 14468.
Buffer Size : 1000B 512.
Handler : RK:
Device Info :

Random Access Device

Record Size in bytes : 1B 1.
Pointer to End of Valid Data : 34204B 14468.
Records Per Block : 1000B 512.
Terminal # : 0
Last I/O Error : 27

Error occurred at line 10 in program cover

Error Termination Status

Both the Pascal-2 compiler and Pascal programs return a termination status when they exit. The Pascal-2 compiler terminates with a "severe error" status if it detects compilation errors. Upon detecting an error while running, such as "subscript out of bounds," a Pascal program also will terminate with a "severe error" status. Otherwise, a "successful completion" status is returned.

The termination status can be used by the command file processor and the batch processor to terminate a command stream that encounters an error. For instance, a command file that compiles and links a Pascal program can use the compiler termination status to detect any errors and skip the link step.

The `Exitst` procedure is a support library routine that sets the termination status of an executing program when a "severe error" status is detected. The procedure's integer argument determines the termination status for any program that calls it. When a "severe error" status of 4 is passed, the procedure also invokes the post-mortem analyzer to create a walkback of the program execution from the point of failure. `Exitst` takes its integer argument, as shown:

```
procedure Exitst(Status: integer);      { procedure declaration }
external;
```

Call the procedure at a point in the program where you want to exit in case of a severe error, as shown:

```
begin      { program Severe }
:
Exitst(4); _____ terminate with severe status
:
end.      { program Severe }
```

A status of 1 means normal termination; any other status means that an error terminated the program.

Implementation Notes

Multiple Source Files

To combine multiple Pascal-2 files into a single compilation unit, you may use multiple input files on the compilation command line, the `%include` extended language feature within the program text, or both.

The choice depends on the need. If, for instance, you are preparing programs for different machines, you can separate machine-dependent data from your individual programs and use the configuration data in a "header" file on the compilation command line.

The `%include` directive allows the inclusion of separate text files within a program, thus simplifying the calling of external procedures. The directive is written as:

```
%include 'file-name-string';
```

The contents of the file specified by *file-name-string* are inserted at the point of the `%include` directive. The string must contain at least the name of the file; if no file extension is specified, `.PAS` is assumed. In addition to the file name and extension, *file-name-string* can contain such information as the logical device name and disk volume number of the file.

The single quotes ('...') enclosing *file-name-string* are optional and provide portability to other implementations of Pascal-2. Despite their optional nature, we recommend that you use the single-quote delimiters on all `%include` directives.

Examples:

```
%include 'hdr';  
- %include 'sy:string';  
%include 'dk0:libdef.pas';
```

Each included file may itself contain `%include` directives, to a maximum nesting of seven levels.

The example below illustrates the use of both header files and the `%include` directive.

Assume that the source file `CONFIG` consists of this:

```
{ This file contains configuration data that is }  
{ subject to change from installation to installation. }  
  
const  
  MaxEntries = 10;      {entries allowed}  
  Debug = false;       {if true, make debugging calls}
```

Assume also that the source file `COMDEF` consists of this:

```
{ This file contains the definitions of some external }  
{ procedures, together with the type declarations needed }  
{ by the main program and the external routines. }  
  
const  
  NameSize = 24;        {size of name field}  
  
type  
  DataItem = record      {describes a customer}  
    Name: packed array [1..NameSize] of char;  
    Age: 0..maxint  
  end;
```

1875

...

...

...

...

...


```

procedure ReadData(var ThisItem: DataItem; {result of read}
                  var Done: boolean {No more items} );
    external;

procedure WriteData(ThisItem: DataItem {item to write} );
    external;

```

And assume that the source file EXAMPL consists of this:

```

%include 'comdef';

var
    Base: array [1..MaxEntries] of DataItem;
    Buf: DataItem;

    Counter: 0..MaxEntries; {count of items in data base}
    I: 0..MaxEntries;       {induction var}

    Done: boolean;          {set when no more items}

begin
    Counter := 0;
    repeat
        ReadData(Buf, Done);
        if not Done then begin
            Counter := Counter + 1;
            Base[Counter] := Buf;
        end;
    until Done;

    { Process data base }

    for I := 1 to Counter do
        WriteData(Base[I]);
    end.

```

These files are compiled with the command:

```

.R PASCAL
*CONFIG,EXAMPL

```

The result is an object module, EXAMPL.OBJ, containing the output from the compilation of CONFIG, COMDEF, and EXAMPL, concatenated in that order. The object module can then be processed through the Linker to produce an executable image.

Any compilation switches used will apply to all input files.

Local Files Closed on Procedure Exit

Consider a procedure (or function) that opens one or more files local to that procedure. Assume that the file variable for the file is defined local to that procedure. When the procedure exits and returns to the calling routine, all files defined local to that procedure are closed. This convention is necessary because upon procedure exit all local variables are deallocated. Once local variables are deallocated, they cannot be referenced again. Therefore, if a local file variable is deallocated, your program can no longer access that file, and no other program may access the file until your program terminates.

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE
COMMISSIONER OF THE
BUREAU OF CHEMISTRY

FOR THE YEAR
1911

CHICAGO, ILL.
1912

PRINTED BY THE
UNIVERSITY OF CHICAGO PRESS

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE
COMMISSIONER OF THE
BUREAU OF CHEMISTRY

FOR THE YEAR
1911
CHICAGO, ILL.
1912

Pascal-2 V2.1/RT-11 Programmer's Guide

To prevent files opened in a procedure from being closed upon procedure exit, define the file variable as a global variable. Since the file variable is in the global data area, the file remains open and accessible until the file is explicitly closed or the program terminates.

Specifying the Location of the Compiler's Work Files

The Pascal-2 compiler opens several temporary scratch files when it compiles a program. For large Pascal programs these files can become quite large (several hundred blocks) and they can be used quite heavily. The V2.1 compiler will attempt to open its scratch files on the logical device called **WK:**. If this logical device does not exist, the scratch files are opened on **SY:**, the system device.

If you are running on a multi-disk system, and the disk you are using has very little free space, you can assign **WK:** to some other disk that has more room. Use the **ASSIGN** command to do this, as shown below:

```
.ASSIGN DL1 WK
```

This command associates the disk **DL1:** with the logical name **WK:**. The compiler then opens its scratch files on **DL1:**.

If your system has different kinds of disks, you should assign **WK:** to the fastest disk on your system. This will reduce compilation time for large programs. You can experiment by using the **times** compilation switch to see if there is a significant change in compilation times with various disks on your system.

Variable Initialization

The Pascal standard states that variables must be initialized before they are used. Otherwise, their values are unpredictable. Pascal-2 catches most uninitialized variables but can't possibly flag all of them. In short, variable initialization is the programmer's responsibility.

Obvious cases are easily detected, but more complex violations such as the initialization of **I** below are not caught by the compiler.

```
var
  I, X: integer;

begin
  read(X);
  if X <= 0
    then I := 0 _____ variable is initialized here
    else I := I + 1; _____ but not here
end.
```

Reading Command Lines

To prompt for and read the command line from an interactive program, simply write the prompt and read the command line, as shown in the following simple example:

```
var
  CommandLine: packed array [1..80] of char;

begin
  write('*');
  readln(CommandLine);
  writeln(CommandLine);
end.
```


However, if you execute the above program as a batch job or an indirect command file, the RT-11 batch processor still expects the command line to be entered from the terminal. This can defeat the whole purpose of running batch or indirect command files; you may want to be free to do other things while the batch job is executing. In this situation, the command line can just as easily be contained in the batch or command file.

With the use of the support library procedure `getlin`, your program can read the command line from the batch processor as well as from the terminal. This flexibility allows you to write programs that can be executed in batch mode or interactively using the same method of obtaining the command line.

The entry point `getlin` is defined in the support library in `OPTRAP.MAC`, a collection of run-time trap routines and entry points that must be in the root segment of any Pascal program that uses overlays. `Getlin` is declared as an external procedure.

The call to `getlin` involves two entry points. The first entry point is `getlin` itself, which simply jumps to the second entry point, `p$gtln`. `P$gtln`, which is contained in the support library module `OPGTLN.MAC`, then writes the '*' prompt and reads the command line. From `p$gtln`, control returns to the module that originally called `getlin`.

The use of dual entry points to read a command line provides the ability to overlay the actual `p$gtln` code against the calling program without overlay conflicts. In this way you can process the command line in the first overlay segment, with the rest of the program overlaid in successive segments.

NOTE

Though `p$gtln` is a separate entry point callable from `getlin`, we recommend that you avoid calling `p$gtln` directly. The results can be unpredictable.

`Getlin` and its required parameters and data types must be declared in your program similar to the following:

```

type
  CmdIndex = 1..CmdLen;
  CmdBuffer = packed array [CmdIndex] of char;

var
  Cbuff: CmdBuffer;
  Cblen: CmdIndex;

procedure getlin(var Cbuff: CmdBuffer; { resulting line }
                 var Cblen: CmdIndex); { length of line }
external;
```

where

- `CmdLen` is the maximum length of the command line. The length of the command line should be at least 80 characters to allow for any command line up to that length.
- `Cbuff` is the packed array of characters into which the command line is read. Spaces may be used in the command line but are stripped off during the read.
- `Cblen` is the length of the command line in bytes. If `Cblen` is returned zero, a command line is not present.

The support library passes the parameters to `getlin` through a temporary storage area, `P$AREA`. In this way both local and global variables can be passed to `getlin` without restriction. In addition

1890

Received of the Treasurer of the County of ... the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

the sum of ...

for the purchase of ...

Pascal-2 V2.1/RT-11 Programmer's Guide

to protecting the command line from being trashed, P\$AREA alleviates memory overwrite problems caused by the USR's swapping in and out of memory to fulfill a request.

The following sample program, FPRINT.PAS, shows a way to check for and read a file name specified in a command line. After it has read the file name, the program simply prints the contents of the file.

```
program FilePrint;

const
  CmdLen = 80;

type
  CmdIndex = 1..CmdLen;
  CmdBuffer = packed array [CmdIndex] of char;

var
  Cbuff: CmdBuffer;
  Cblen: CmdIndex;
  Filename: packed array [1..CmdLen] of char;
  Ch: char;
  I: integer;

procedure getlin(var Cbuff: CmdBuffer; { resulting line }
                 var Cblen: CmdIndex); { length of line }
external;

begin
  getlin(Cbuff, Cblen);           { make command line available }
  if Cblen <> 0 then begin        { command line present }
    for I := 1 to Cblen do
      Filename[I] := Cbuff[I];
    reset(input,Filename);       { open the input file }
    while not eof do begin       { once per line }
      while not eoln do begin    { once per char }
        read(Ch); write(Ch);
      end;
      readln; writeln;
    end;
  end
  else writeln('Err -- No file name given');
end.
```

Compile and link the above program with these steps:

```
.R PASCAL
*FPRINT
.LINK FPRINT,SY:PASCAL
```

The program then can be executed interactively or indirectly with the same results. Below, the

1870

1871

1872

1873

1874

1875

1876

1877

1878

1879

1880

the contents of CODE.DAT is directed to the terminal.

```
.RUN FPRINT _____ interactive execution
*CODE.DAT
:
{ contents of CODE.DAT }
:
:
```

For indirect execution, place the command and command line in a separate file with a .COM extension. FPRINT.COM could contain the following lines:

```
RUN FPRINT _____ the command
CODE.DAT _____ the command line
```

The at-sign indirect command symbol is used to execute the command file.

```
@ FPRINT
.RUN FPRINT
*CODE.DAT
:
{ contents of CODE.DAT }
:
:
```

NOTE

A possible drawback to the use of `getlin` is slower program execution, especially on floppy-disk systems that allow the `USR` to swap.

Foreground Operation

For foreground operation, allocate additional memory to ensure sufficient space for the stack, the heap, and file buffers. Each Pascal file requires about 300 words (more for large buffers), so allocate at least 600 words for the default input and output files. Use the `/BUFFER:` switch, as shown below. The period after 1024 signifies an octal value.

```
.FRUN <file-name>/BUFFER:1024.
```

1. 1. 1.

2. 2. 2.

3. 3. 3.

4. 4. 4.

5. 5. 5.

6. 6. 6.

7. 7. 7.

8. 8. 8.

9. 9. 9.

10. 10. 10.

11. 11. 11.

Lazy I/O

For standard Pascal, an interactive input file, such as terminal input, poses a problem. A program must always be able to determine the current status of an open file, i.e., it must be able to retrieve the current record from the buffer variable (F⁻) and the current values of `eof` and `eof`. Since an interactive file is being created as it is being read, the program must periodically wait for you to enter a full line before it can determine the "end of line" (or the "end of file") and keep the file well-defined. In this way the program is not synchronized with interactive input.

Pascal-2 uses an input interface known as "lazy I/O" to handle input from text files. Since a file's status needs to be defined only when the program actually refers to it, lazy I/O can safely delay any input operation until the program uses its results. When a Pascal program requests an input operation on a text file, the operation is recorded for later use. The delayed operation is triggered by any subsequent reference to the file's buffer variable, the `eof` or the `eof` value. The delay is invisible to the program but is visible to the user from the way the program is synchronized with interactive input.

To use lazy I/O, you need to be aware of its effect on synchronization of input and output operations. As an example, consider a simple program that reads its standard file input, which is connected to a terminal. The program prompts for each line, and stops at the end of the file. The design of the program is dictated by two requirements:

1. For the prompt to be effective, it must appear before the user is required to type the line.
2. To detect the end of the file correctly, the program must check for it before reading each line.

To meet both of these requirements, the program must print the prompt before an operation is performed that requires the next line of the file to be known: checking for "end of file" or reading the line. The following example shows this design.

```
program Interactive(input, output);  
  
begin  
  write('prompt:');  
  while not eof do  
    begin  
      readln;  
      write('prompt:');  
    end  
  end.  
end.
```


Incorporating Lazy I/O into V3.0 Programs

Since lazy I/O changes the way Pascal-2 performs interactive I/O, programs compiled with Pascal-2 Version 2.0 may have to be revised so they conform to the new scheme. For example, the 2.0 version of the above program might have been coded as follows:

```

program Interactive(input, output);

begin
  while not eof do
    begin
      write('prompt:');
      readln;
    end
  end.

```

If you compile this program and execute it, the program appears to "hang." In reality, the program is waiting for input; it is attempting to define the value of `eof` in the `while` statement, before it has the chance to write the prompt. Unfortunately, you will never see the prompt until you type in a line. This means that you will be prompted for the line you have just entered. In this case the program is "out of sync" with input from the terminal.

Terminal I/O

The RT-11 terminal driver acts as an intermediary between a Pascal program and the video terminal from which it is running. The terminal driver, under operating system direction, collects characters one at a time and at end of line, sends a full line of text to the screen (for a `writeln`) or to the executing program (for a `readln`).

For example, when a `read` statement reads input from a terminal, the terminal driver reads each character and places it in an internal buffer until the terminal driver encounters an end of line (a carriage return, line feed, escape or Control-Z ^Z). At this point, the terminal driver sends the entire line to the program and lets the program process the line one character at a time.

The opposite is true for `write` statements, where no buffering is performed and each character is displayed on the terminal at the time the program writes it.

As previously mentioned, the `writeln` statement directs the terminal driver to write the rest of the current line and prepare for the next line of output. The usual output sequence sent by the terminal driver is: data, carriage return, line feed. This sequence prints the data, returns the cursor to the first position of the newly printed line and moves the cursor to the beginning of the next line.

The normal sequence of commands issued by the terminal driver may not particularly suit special I/O applications such as direct cursor addressing or special function key processing. To gain control of the terminal-I/O command sequence, use the `/odt` file control switch in conjunction with the `/bufferize:n` switch on the `reset` statements. Used together, the `/odt` and `/bufferize:1` switches allow your program to read terminal input one character at a time without the need for a line terminator. To recognize special-function keys such as the PF1 ("gold") key on VT100 keyboards, use the `readsa` procedure available in the Pascal support library. Both of these features are described in the following sections.

Handwritten header or title at the top left of the page.

First main paragraph of text, starting with a capital letter, located in the upper middle section of the page.

Second main paragraph of text, continuing the narrative or discussion, located below the first paragraph.

Third main paragraph of text, located in the middle section of the page.

Fourth main paragraph of text, located below the middle section.

Fifth main paragraph of text, located in the lower middle section.

Sixth main paragraph of text, located below the lower middle section.

Seventh main paragraph of text, located in the lower section of the page.

Eighth main paragraph of text, located near the bottom of the page.

Ninth main paragraph of text, located at the very bottom of the page.

Single-Character or 'ODT' Mode

On RT-11, single-character input between a video terminal and an executing Pascal program is accomplished with the use of the `/odt` and `/buffersize:1` file control switches on the `reset` statement that opens `II:` as the input file (treated as a text file).

When the `/odt` file control switch is specified, each character read from the keyboard processed immediately without the program waiting for a carriage return or other action character. The `/buffersize:1` switch sets the terminal's buffer size to 1 and places the device in single-character mode.

Example:

```
reset(input, 't1:/odt/buffersize:1');
```

A `reset` statement having these switches creates a new file control block and buffer and redirects all references to standard input to the terminal, known to the support library as `II:`. Consequently, the library calls the system routine `.TTYIN` rather than `.READ` to perform the terminal I/O. `Reset` overhead is minimal because no actual `.LOOKUP` is performed to open the file.

In single-character or "odt" mode, a program reads input from the terminal one character at a time without waiting for a line terminator (i.e., the line-feed and escape characters). As a result, the programmer is responsible for echoing each character as it is read.

NOTE

Though not a line terminator *per se*, the `RETURN` key terminates a line of input because it performs a carriage return then a line feed, which terminates the line. The null and carriage-return characters are ignored.

When line-feed and escape characters are encountered, the `eoln` flag is set to true and `input` points to a space. Likewise, detection of Control-Z (^Z) or the physical end of file sets `eof` to true and `input` to a space.


```

program Single;
var
  Ch: char;

begin { Single }
  reset(input, 'ti:/odt/buffersize:1');
  repeat { Forever -- end it with Control-C }
    write('Prompt >');
    while not eoln do begin
      read(Ch);
      write(Ch);
    end; { while }
    readln;
    writeln;
  until false;
end. { Single }

```

Program **Single** prompts for input from the terminal. The characters you type appear to be echoed to the screen normally as they are typed. However, the program, and not the operating system, is doing the echoing (the `write(Ch)` statement).

Function 'ReadSn'

A second method for reading a single character from the terminal (a text file) entails the use of the Pascal support library function `readsn`. In addition to reading a single character at a time, `readsn` allows your program to recognize and correctly interpret the special function keys that are available on many terminals.

`Readsn` has the same effect as "odt" mode with a one-byte buffer size and no character echo. `Readsn` does none of the normal text-file processing done by the support library, in which escape characters (<ESC>) are stripped from the file. Since `readsn` does not require a special buffer or file control variable, `reads` and `readlns` from standard input still produce the expected results. You must, however, buffer the characters internally in your program if you wish to save the characters that are typed.

Declare the function as an external, as shown:

```
function ReadSn: char; external;
```

The returned value of `readsn` is the current character read from the terminal.

Special-function keys send a sequence of characters that is interpreted as one key. An example of special-function keys is the top row of keys on the VT100's 10-key keypad marked PF1 to PF4. With the use of `readsn`, the sequence of characters making up special-function keys can be picked off one character at a time. For example, the PF1 key is made up of the three-character sequence '<ESC>OP' (<ESC>, O and P). By reading the sequence one character at a time using `readsn`, a program can determine that the PF1 key was pressed and not a carriage-return or line-feed key followed by 'O' then 'P.'

1. The first part of the document is a letter from the President of the United States to the Congress, dated January 3, 1862.

2. The second part of the document is a report from the Secretary of the Treasury, dated January 3, 1862.

3. The third part of the document is a report from the Secretary of the Interior, dated January 3, 1862.

4. The fourth part of the document is a report from the Secretary of the Navy, dated January 3, 1862.

5. The fifth part of the document is a report from the Secretary of the War, dated January 3, 1862.

6. The sixth part of the document is a report from the Secretary of the State, dated January 3, 1862.

7. The seventh part of the document is a report from the Secretary of the War, dated January 3, 1862.

8. The eighth part of the document is a report from the Secretary of the Navy, dated January 3, 1862.

9. The ninth part of the document is a report from the Secretary of the Interior, dated January 3, 1862.

10. The tenth part of the document is a report from the Secretary of the Treasury, dated January 3, 1862.

11. The eleventh part of the document is a report from the Secretary of the War, dated January 3, 1862.

12. The twelfth part of the document is a report from the Secretary of the Navy, dated January 3, 1862.

13. The thirteenth part of the document is a report from the Secretary of the Interior, dated January 3, 1862.

14. The fourteenth part of the document is a report from the Secretary of the Treasury, dated January 3, 1862.

15. The fifteenth part of the document is a report from the Secretary of the War, dated January 3, 1862.

Random Access to 'Text' Files

The `seek` procedure cannot compute the location of a particular record (line) within a file of type `text` because the lines are of variable lengths. The Pascal support library supplies two external procedures, `getpos` and `setpos`, that simulate random access to text files.

These two procedures are not predefined and must be declared in your program as `external`. `Getpos` determines the starting location of the next line of a file, and `setpos` sets the file pointer to the specified starting location of a line within the file. The beginning of each line of a text file is denoted by a block number and a byte offset into that block. Each block contains 512 bytes. The first line of a file starts with block 1, offset 0. If you try to access a nonexistent position or a position in the middle of a line, an I/O error will result.

The block number and byte offset must be values returned by `getpos`. You cannot compute the values yourself. When the file is being read, use `getpos` to determine the starting position of the next line and save that block and offset combination for later use by `setpos`.

Bear in mind that this is not "true" random access; you cannot access individual characters, only individual lines of text. Use your Pascal program to access characters individually within each line.

Procedure 'GetPos'

Procedure `getpos` determines the starting position of the next line to be read from or written to a text file. `Getpos` requires three parameters, passed by reference, as shown below:

```
procedure GetPos(var F: text; var Block, Offset: integer);
  external;
```

where

F is the file variable of type `text`.

Block is the returned disk block number of the next line in file *F* to be read or written.

Offset is the returned byte offset into *Block*. Together, *Block* and *Offset* point to the next line to be processed.

You should always call `getpos` to obtain the location in the file before you call `setpos`, so the block and offset values being passed to `setpos` are valid.

The example in the next subsection shows the use of `getpos`.

1914

1915

1916

1917

1918

1919

1920

1921

1922

Procedure 'SetPos'

Procedure **setpos** positions the file pointer to a specified block number and byte offset into that block. **Setpos** accepts the same three parameters as **getpos**, except *Block* and *Offset* are passed by value. The **setpos** declaration is as follows:

```
procedure SetPos(var F: text; Block, Offset: integer);
  external;
```

where

F is the file variable of type **text**.

Block is the block number to which the file pointer is set.

Offset is the byte offset into *Block*. Together, *Block* and *Offset* point to the new position.

To stress an earlier point, the block number and byte offset must be values returned by **getpos**. Do not attempt to compute the values yourself. Save the returned values for later use.

If an error is detected while **setpos** tries to position the file, the end-of-file flag **eof** is set to true. The **ioerror** and **iostatus** support library functions may help you to determine the reason that the line could not be accessed. (For details on **ioerror** and **iostatus**, see "I/O Error Trapping" in this section.) If a file is positioned to a block and offset that does not correspond to the first character of a line, the results are unpredictable.

The example below shows a way to use **getpos** and **setpos**. Program **Reverse** reads a text file and saves the position of each line in a linked list. It then prints the file in reverse line order so that the last line of the file is printed first and the first line is printed last.


```

program Reverse;

type
  Pointer = ^position;
  position =
    record
      Next: pointer;
      Block: integer;
      Offset: integer;
    end;

var
  F: text;
  Filename: packed array [1..60] of char;
  P, X: pointer;
  Done: boolean;

procedure GetPos(var F: text; var Block, Offset: integer);
  external;

procedure SetPos(var F: text; Block, Offset: integer);
  external;

begin
  write('File name? ');
  readln(Filename);
  reset(F, Filename);
  P := nil;
  repeat
    { read the file }
    new(X);
    with X do GetPos(F, Block, Offset);
    X.Next := P;
    P := X;
    Done := eof(F);
    if not Done then readln(F);
  until Done;

  while P <> nil do
    { write the file }
    with P do begin
      SetPos(F, Block, Offset);
      if not eof(F) then begin
        while not eoln(F) do begin
          write(F);
          get(F);
        end;
        writeln;
      end;
      P := P.Next;
    end;
  end.

```

Handwritten text at the top of the page, possibly a title or header.

Handwritten text in the upper middle section.

Handwritten text in the middle section, possibly a list or series of notes.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Unsigned Integer Conversion

On the PDP-11, integer variables are stored in 16-bit words. These 16-bit words may be interpreted as signed or unsigned integers. A signed number, in two's complement notation, represents numbers in the range $-32767..32767$. An "unsigned" (also called "extended-range") number by definition does not have a sign bit; rather, it uses all 16 bits to represent an integer in the range $0..65535$.

When their values are compared or used in mathematical expressions, unsigned integers differ greatly from signed integers. As an example, consider a word in which all 16 bits are set to one. This word has a value of -1 when interpreted as a signed integer, or a value of 65535 when interpreted as an unsigned integer. When this word is compared with some other value, the PDP-11 uses different combinations of instructions for signed and unsigned comparisons. If this number is multiplied by two, the result is a value of -2 for signed or 131070 for unsigned. The latter is an overflow condition because the result does not fit within 16 bits.

The Pascal-2 compiler and support library also differ in their treatment of signed and unsigned integers. When you define a variable to be of type `integer` in your Pascal program, the compiler treats that value as a signed integer, unless you specify an unsigned integer using a subrange notation such as:

```
type
  unsigned = 0..65535;
```

```
var
  X: unsigned;
```

According to your data declarations, the compiler will generate the correct code to compare, multiply, or divide unsigned numbers. The compiler can then deal with unsigned integers.

However, if you attempt to write out the value of an unsigned integer, you will find that the number is always treated as a signed integer. This occurs because the Pascal support library uses a single routine to print integers. This routine interprets all integers as signed values. If you want to write out the value of an unsigned integer, use the following procedure in your program instead of the `write` statement. This procedure, `Uwrite`, takes an unsigned integer and a field width as arguments. The number is printed as a value in the range $0..65535$, right justified in the specified field.

```
procedure Uwrite(X: unsigned;
  Width: integer);

{ This procedure writes an unsigned integer to output. }

begin { Uwrite }
  if (X > 32767) and (Width >= 0) then
    begin
      if Width > 0 then
        Width := Width - 1;
      write(X div 10: Width);
      X := X mod 10;
      Width := 1;
    end;
  write(X: Width);
end; { Uwrite }
```

The PDP-11 floating-point hardware uses signed conversion when it converts an integer value to a real value. If you wish to convert an unsigned integer to real, use the following function. This

Handwritten text, likely a letter or document, covering the upper half of the page. The text is written in cursive and is mostly illegible due to fading.

Handwritten text, likely a letter or document, covering the middle section of the page. The text is written in cursive and is mostly illegible due to fading.

Handwritten text, likely a letter or document, covering the lower middle section of the page. The text is written in cursive and is mostly illegible due to fading.

Handwritten text, likely a letter or document, covering the bottom section of the page. The text is written in cursive and is mostly illegible due to fading.

Pascal-2 V2.1/RT-11 Programmer's Guide

function, Ufloat, takes an unsigned integer as its argument and returns a real value in the range of 0.0 to 65535.0.

```
function Ufloat(X: unsigned): real;

  { This function converts an unsigned number to a real number. }

var
  R: real;

begin { Ufloat }
  R := X;
  if R < 0.0
    then R := R + 65536.0;
  Ufloat := R;
end; { Ufloat }
```

The trunc and round functions convert real numbers to integers. Since the floating-point hardware assumes a signed conversion, the following function should be used when an unsigned integer result is desired. The function Utrunc takes a real number in the range 0.0 to 65535.0 and converts it to an unsigned integer.

```
function Utrunc(R: real): unsigned;

  { This function converts a real number to an unsigned integer. }

begin { Utrunc }
  if (R > 65535.0) or (R < 0.0)
    then writeln('Unsigned number out of range');
  if R > 32767.0
    then R := R - 65536.0;
  Utrunc := trunc(R);
end; { Utrunc }
```

The unsigned round function is very similar to the above unsigned trunc function.

My dear Sir,
I have the honor to acknowledge the receipt of your letter of the 10th inst. and in reply to inform you that the same has been forwarded to the proper authorities for their consideration.

I am, Sir, very respectfully,
Your obedient servant,
J. H. [Name]

I am, Sir, very respectfully,
Your obedient servant,
J. H. [Name]

I am, Sir, very respectfully,
Your obedient servant,
J. H. [Name]

Compiler Optimizations

The Pascal-2 compiler implements these optimizations:

Variable Assignments to Registers

The compiler permanently assigns up to three floating-point accumulators and two general registers to commonly used local variables in each block. The compiler assigns the registers to the variables that are the most often used. No register is assigned for variables passed to a procedure as a **var** parameter or referenced directly by a procedure local to the declaring procedure. In addition, this optimization is disabled for the main program if any external procedures are referenced, since the compiler cannot determine what variables may be used by such routines.

Assignment of Constants and Addresses to Registers

The compiler attempts to fill all registers with useful operands during compilation of a procedure, since operations on registers are faster and take less space than the corresponding operation performed in memory. Once a procedure is compiled, unused registers are filled with constant operands and addresses if such assignment saves space. This low-level optimization often results in a saving in execution time as well.

Constant Folding

The compiler directly evaluates (folds) simple arithmetic involving constant operands of the types **integer**, **char**, **real**, and **boolean**. The generated code contains the result rather than the expression. (The RT-11 SJ compiler does not fold **real** constants; the XM compiler does.) Set expressions and relational expressions are not folded.

Dead Code Elimination

If statements and **case** statements are optimized if the selection expression is constant. In this case only one path of execution is possible, and the compiler discards others. Knowledge of this optimization can lead to the writing of conditional code much like that available in some preprocessors. For example:

```
if Debugging then writeln(SomeUserValue);
```

No code for this statement is generated if the identifier **Debugging** is defined as a constant with the value **false**.

The **debug** compilation switch disables this optimization.

Boolean Expression Optimization

When appropriate, Pascal-2 uses the minimum number of operations necessary to compute the final value of operands in Boolean expressions, thereby reducing the cost of evaluating individual Boolean expressions. (This method is known as a "short-circuit" evaluation.) The programmer must be careful not to assume that all operands of Boolean operators are evaluated or that some may not be evaluated. (This optimization takes advantage of a provision in the Pascal standard that allows an implementation to evaluate only the necessary operands of a Boolean expression.) Also, the order in which the operands are evaluated is unpredictable.

Handwritten text at the top left of the page.

Handwritten text in the upper middle section of the page.

Handwritten text in the upper right section of the page.

Handwritten text in the middle section of the page.

Handwritten text in the lower middle section of the page.

Handwritten text in the lower section of the page.

Handwritten text in the lower section of the page.

Handwritten text in the lower section of the page.

Handwritten text in the lower section of the page.

Handwritten text in the lower section of the page.

Expression Targeting

The compiler can determine from context where a particular expression result should be computed. For instance, procedure parameters can often be computed directly on the run-time stack, and at times, expressions on the right side of the assignment operator can be computed directly into the variable on the left side.

Common Subexpression Elimination

Multiple occurrences of the same expression are detected and simplified. Such optimization of redundant expressions is needed even though a programmer can often avoid writing such code by introducing auxiliary variables. For instance, this example:

```
writeln(I + 1, I + 1);
```

may be simplified to:

```
J := I + 1; writeln(J, J);
```

The simplification avoids the redundant computation. However, redundancy of the sort shown in the first example often leads to a more readable program. Also, certain classes of redundant expressions cannot be eliminated in the source program. For instance, array index calculations involve several underlying operations that are not reflected in the source code and therefore cannot be simplified by the programmer. Pascal-2 eliminates a wide class of common subexpressions, across statement boundaries as well as within simple expressions.

The `debug` compilation switch disables this optimization.

Common Branch Tail Elimination

In some cases the compiler generates several branches to the same location in the object program. At times the compiler can replace redundant instructions preceding one such branch instruction with a branch to a point in the generated code that executes the same instruction stream. This low-level optimization executes an extra branch instruction in order to save some space.

The `debug` compilation switch disables this optimization.

Array Index Simplification

Index expressions of the form `[variable + constant]` and `[variable - constant]` are partially computed. The addition or subtraction of the constant operand is folded into the value computed for the base of the array. This optimization is enabled **only** if array bounds checking is disabled and the array is unpacked.

1. The purpose of this document is to provide information regarding the security of the system. It is intended for use by personnel who are responsible for the operation and maintenance of the system.

2. The information contained in this document is classified as CONFIDENTIAL - SECURITY INFORMATION. It is to be controlled and handled in accordance with the applicable security policies and procedures.

3. The information contained in this document is to be used only for the purpose for which it was provided. It is not to be disseminated to other personnel unless authorized in writing by the appropriate authority.

4. The information contained in this document is to be stored and handled in accordance with the applicable security policies and procedures. It is to be protected from unauthorized access, use, and disclosure.

5. The information contained in this document is to be destroyed when it is no longer needed for the purpose for which it was provided. It is to be destroyed in accordance with the applicable security policies and procedures.

6. The information contained in this document is to be controlled and handled in accordance with the applicable security policies and procedures. It is to be protected from unauthorized access, use, and disclosure.

Appendix A: Compilation Error Messages

'(' expected
Check parameter list syntax.

)' expected
Check parameter list syntax.

',' expected
Check parameter list syntax.

'...' expected
Check array specification.

':' expected
Check type or var specification.

':'= expected
Check for undefined procedure or missing colon.

';' expected after procedure body
Use semicolons to separate procedure declarations.

'=' expected
Check constant or type syntax.

'[' expected
Check array index specification.

']' expected
Check array index or set specification.

']' or ', ' must follow index expression
Check array index specification.

A TYPE identifier is not allowed here
The compiler encountered a bad structured constant or misplaced identifier.

Actual parameter cannot be used with this conformant array parameter
When conformant array parameters are used, the actual parameter must be an array and its type must be compatible with the formal parameter type.

Actual parameter type doesn't match formal parameter type
Parameters being passed to procedures must have the same type names as the declared (formal) parameters.

All parameters in a single parameter section must have the same type
Parameters grouped under one conformant array schema must be of the same type.

Ambiguous switch
The specified command-line switch name does not contain enough characters to distinguish it from switches with similar names.

Pascal-2 V2.1/RT-11 Programmer's Guide

Array exceeds addressable memory

Array subscript out of range

Assignment of file variables not allowed

An attempt has been made to assign an expression to a file variable or one file variable to another.

Assignment operands are of differing or incompatible types

Type mismatch — compare left and right sides of assignment statement for compatibility. Note that pointer types must point to identical data structures.

Assignment to constants not allowed

Assignment value out of range

Bad adjust offset value in procedure <name>/main program

A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

Bad CASE label

Case labels and case selectors must be of the same type. A colon, erroneously placed after the keyword **otherwise**, can also cause this error.

Bad IN operands

The left operand must be of a scalar type; the right operand must be of a compatible set type.

Bad ORIGIN value

Origin values are restricted to the I/O page (locations 0 to 1000 octal) or the system area (28K to 32K).

Bad constant

Bad file name syntax

Check command-line syntax.

Bad parameter element

The indicated parameter element was not followed by a **'** or a **)**.

Bad type syntax

Badly formed expression

Check parentheses and operator placement.

BEGIN expected

The statement part of a block must start with **begin**. Modules with no main program require the **nomain** compilation switch.

Binary operator expected

Two operands must be separated by an operator. Also check for mismatched quotes or an illegal digit in the cross-hatch **#** integer form.

Appendix A: Compilation Error Messages

Block declarations are incorrectly ordered

The relaxed ordering of declarations is an extension to standard Pascal and may be used only for global declarations.

Block ended incorrectly

Block must begin with LABEL, CONST, TYPE, VAR, PROCEDURE, FUNCTION, or BEGIN

Boolean value expected

Can't assign a real value to an integer variable (use TRUNC or ROUND)

Can't pack unstructured or named type

CASE label defined twice

CASE label does not match selection expression type

CASE label must be non-real scalar type

CASE label type does not match tag field type

CASE selection expression must be a non-real scalar type

Code too complex in procedure <name>/main program

The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Compiler writer error -- please contact Oregon Software at (503) 228-7760

This indicates an internal compiler error — please save all listings and terminal output.

Conflicting switches specified

Certain switch combinations cannot be specified together on the compiler command line. The debug switch conflicts with both the profile and errors switches; the profile switch conflicts with the errors switch; and the object switch conflicts with the macro switch.

Declaration terminated incorrectly

Declared labels must be defined in procedure body

DO expected

Check for, while or with statement syntax.

END expected

Exponent must lie in range -38..38

Expression type is incompatible with FOR index type

For statement index types must be non-real scalars.

External procedures/functions must be defined at outermost level

External procedures may not be defined within other procedures.

Handwritten text at the top of the page, possibly a title or header.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Fifth main paragraph of handwritten text.

Sixth main paragraph of handwritten text.

Seventh main paragraph of handwritten text.

Eighth main paragraph of handwritten text.

Ninth main paragraph of handwritten text.

Tenth main paragraph of handwritten text.

Eleventh main paragraph of handwritten text.

Twelfth main paragraph of handwritten text.

Thirteenth main paragraph of handwritten text.

Fourteenth main paragraph of handwritten text.

Fifteenth main paragraph of handwritten text.

Sixteenth main paragraph of handwritten text.

Seventeenth main paragraph of handwritten text.

Eighteenth main paragraph of handwritten text.

Nineteenth main paragraph of handwritten text.

Twentieth main paragraph of handwritten text.

Twenty-first main paragraph of handwritten text.

Twenty-second main paragraph of handwritten text.

Twenty-third main paragraph of handwritten text.

Twenty-fourth main paragraph of handwritten text.

Pascal-2 V2.1/RT-11 Programmer's Guide

Extra END following block -- Check BEGIN ... END pairing

Extra procedures found after main program body

This error occurs when more than one main program body (starting with a program statement) appears in the source file.

Extra statements found after end of program

This error occurs when more than one main program body appears in the source file, or when the `nomain` compilation switch is used with a source file that contains a main program body.

Field variable expected for NEW

Additional parameters to `new` must be tag-field constant values that identify the particular variant record being allocated.

File cannot contain a file component

An element of a file cannot itself contain a file.

File names in RESET/REWRITE are non-standard

This error is generated only when the `standard` compilation switch is enabled.

File variable expected

The first parameter to `reset`, `rewrite`, `get`, `put`, and `seek` must be a file variable.

File variable or pointer variable expected

The indicated caret (^) has been incorrectly placed after a variable that was neither a pointer nor a file.

Files must be passed as VAR parameters

FOR-loop control variable can only be a simple non-real scalar variable

FOR-loop control variable must be declared at this level

A `for` statement control variable must be declared local to the block containing the `for` statement.

Format expression must be of type INTEGER

Field-width specifications in `write` or `writeln` statements must be integers.

Forward procedure/function body is never defined

Forward type reference is never resolved

The type referenced in a pointer type declaration is not defined by later declarations.

Function cannot be applied to an operand of this type

A standard function has been passed a parameter of the wrong type (for example, `trunc/round` can only be applied to real types).

Function identifier is never assigned a value

Function name expected

1. The first part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

2. The second part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

3. The third part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

4. The fourth part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

5. The fifth part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

6. The sixth part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

7. The seventh part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

8. The eighth part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

9. The ninth part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are derived from the principles of relativity and the theory of the structure of the atom.

Appendix A: Compilation Error Messages

Function result must be of scalar or pointer type

Functions may not return structured types such as records and arrays. Use `var` parameters to do this.

Function result type cannot be duplicated in forward-declared function body

The parameter list and result type are already specified by the forward declaration and may not be repeated. Instead, simply give the function name.

Identifier cannot be redefined or defined after use at this level

The specified identifier is already defined in the current block and cannot be assigned a new meaning in the indicated block.

Identifier expected

The indicated argument should be a variable, not a constant or expression.

Illegal character

Illegal comparison of record, array, file, or pointer values

Pointer types may be compared only for equality; record, array, and file types may not be compared in any case except strings.

Illegal function assignment

Illegal subrange

The lower bound of a subrange is required to be less than or equal to the upper bound.

Index expression type does not match array declaration

Index must be non-real scalar type

Index variable missing in this FOR statement

Integer label expected

Integer overflow or division by zero

Integers must lie in range -32767..32767

Internal temp error in procedure <name>/main program

A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

Label cannot be redefined at this level

Labels may be redefined within nested procedures, but not at the same level.

Label defined twice

Label is target of illegal GOTO

Branching into `if-then-else` or `case` statements is illegal.

Label must be declared in LABEL declaration

My dear Mr. [Name],

I have just received your letter of the 10th inst.

and am glad to hear that you are well.

I am writing you a few lines to let you know

that I have received your letter of the 10th inst.

and am glad to hear that you are well.

I am writing you a few lines to let you know

that I have received your letter of the 10th inst.

and am glad to hear that you are well.

I am writing you a few lines to let you know

that I have received your letter of the 10th inst.

and am glad to hear that you are well.

I am writing you a few lines to let you know

Pascal-2 V2.1/RT-11 Programmer's Guide

Label must be unsigned integer constant

Line too long

The maximum input line length is 180 characters.

Listing requested but no file provided

Check command-line syntax.

More than two output file specifications

Check command-line syntax.

Must assign value before using variable

The standard states that variables must be initialized before they are used.

Must use VAR parameters with NONPASCAL directive

The calling sequence for nonpascal procedures and functions accepts only call-by-reference parameters.

Need at least 1 digit after '.' or 'E'

Check for proper real numeric format.

Need at least one value to WRITE

Need at least one variable to READ

No file in field

Check command-line syntax.

No input file provided

Check command-line syntax.

"NO" not allowed on this switch

No strict inclusion of sets allowed

The operators '<' and '>' may not be applied to set operands. Instead, use '<=' or '>='.

Non-decimal integers are not standard Pascal

This message is issued only when the standard compilation switch is specified and the cross-hatch '#' integer form is used.

Non-standard comment form, please use "{ or "(*"

The comment form '/*', '*/' is not accepted by Pascal-2. The PASMAT utility automatically converts non-standard comments to the standard form.

Nonsense discovered after program end

Extraneous characters are present in the input file after the proper end of the program.

Octal constant contains an illegal digit

Octal constants cannot contain an 8 or a 9.

Appendix A: Compilation Error Messages

Octal constants are not standard Pascal.

This message is issued only when the **standard** compilation switch is specified and the conventional octal form containing 'B' is used.

OF expected

Check file or set declaration syntax, or **case** statement syntax.

Only 15 levels of nesting allowed

The compiler's limit for procedure and function nesting has been exceeded.

Only functions can be called from expressions

Procedures do not return a value and may not be called from within expressions.

Operand expected

Operands are of differing or incompatible type

Operator cannot be applied to these operand types

Check the indicated expression for proper form and operand type compatibility. For example, characters may not be multiplied together.

OTHERWISE/ELSE clause in CASE not allowed

Otherwise is an extension to standard Pascal. This message is issued when the **standard** compilation switch is specified.

Out of memory in procedure <name>/main program

The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Output requested but no file provided

Check command-line syntax.

Packed array [1..n] of characters expected

The file name arguments in **reset** and **rewrite** must be strings.

Packed conformant array parameters cannot be nested

Only one index type specification is allowed for packed conformant array parameters.

Parameter list cannot be duplicated in forward-declared procedure/function body

The indicated statement should simply give the procedure name and no parameters.

Pointer variable expected

Procedure name expected

Procedures cannot be followed by type definition

The relaxation of declaration ordering applies only to global declarations, and to declarations in inner blocks which precede procedure and function definitions. The indicated declaration section is improperly placed.

PROGRAM heading expected

This error occurs only if the **standard** compilation switch is set.

Radix of non-decimal constant must lie in range 2..16

Handwritten text at the top of the page, possibly a title or header.

First paragraph of handwritten text.

Second paragraph of handwritten text.

Third paragraph of handwritten text.

Fourth paragraph of handwritten text.

Fifth paragraph of handwritten text.

Sixth paragraph of handwritten text.

Seventh paragraph of handwritten text.

Eighth paragraph of handwritten text.

Ninth paragraph of handwritten text.

Tenth paragraph of handwritten text.

Eleventh paragraph of handwritten text.

Twelfth paragraph of handwritten text.

Final line of handwritten text at the bottom of the page.

Pascal-2 V2.1/RT-11 Programmer's Guide

READLN and WRITELN must be applied to text file

Reassignment of FOR-loop control variable not allowed

The control variable of a for statement may not be modified inside the body of the for statement.

Record identifier expected

A with statement must specify a record variable.

Required parameter missing

The indicated command-line switch requires a parameter as part of the switch. Consult the section in this manual on compilation switches for the required parameter.

Same switch used twice

Check command line for duplicate switches.

Set is constructed of incompatible types

Set types must have a base in the range 0..255

Sets must be non-real scalar type

The indicated set definition contains an illegal component type.

Statement ended incorrectly

String constants may not include line separator

A closing single quote (') is missing.

String of length zero

Strings must contain at least one character.

Tag does not appear in variant record label list

The tag field referred to does not exist.

Tag identifier already used in this record

Field identifiers within a record are required to be unique and may not be redefined within that record.

The divisor of a MOD must be greater than zero

THEN expected

Check if statement form.

This function was declared as a forward procedure

Conflict between declaration and use of function identifier. Check previous declaration.

This parameter cannot be followed by a format expression

A format expression may appear only in calls to write and writeln.

This procedure was declared as a forward function

Conflict between declaration and use of procedure identifier. Check previous declaration.

This procedure/function name has been previously declared forward

A procedure cannot be both forward and external, or both forward and nonpascal.

Appendix A: Compilation Error Messages

TO or DOWNT0 expected

Check for statement syntax.

Too few actual parameters

The indicated parameter list does not agree with the procedure or function parameter definition.

Too many actual parameters

The indicated parameter list does not agree with the procedure or function parameter definition.

Too many errors!

The compiler error table holds 50 error messages. Error processing is terminated. Correct earlier errors and recompile for further checking.

Too many external references in procedure <name>/main program

Programs are limited to 256 external procedure references. See Appendix C of this guide for more information.

Too many forward references (only 50 allowed)

Too many identifiers (only 1597 allowed)

Too many keys in procedure <name>/main program

The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Too many labels in procedure <name>/main program

The limit of 280 case labels has been exceeded in named body of code. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Too many nested INCLUDE directives (only 8 allowed)

Too many nodes in procedure <name>/main program

The named body of code is too large to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Too many Pascal labels in procedure <name>/main program

More than 32 statement labels have been declared in named body of code. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Too many procedures (only 300 allowed)

Too many strings or identifiers

Restructure the program to reduce its complexity.

Too much object code in procedure <name>/main program

The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

Two file names in one field

Check the command line for missing '=' or ','.

1891

Dear Sir,
I have the honor to acknowledge the receipt of your letter of the 10th inst.

and in reply to inform you that the same has been forwarded to the proper authorities.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

[Address]

[Address]

[Address]

[Address]

[Address]

[Address]

[Address]

[Address]

[Address]

Pascal-2 V2.1/RT-11 Programmer's Guide

Travrs build error in main program

A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

Travrs walk error in main program

A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

Type name expected

The first parameter passed to the `loophole` function must be a type name.

Unable to open compiler scratch files

The disk in use does not have enough free space to store the compiler's scratch files. Try assigning WK: to a different disk.

Unary '+' or '-' cannot be applied to set operands

Undefined identifier

Undeleted temps in procedure <name>/main program

A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

Unexpected ')' -- Check for matching parenthesis

Unexpected ELSE clause -- Check preceding IF for extra ';'

Unknown directive

The legal directives are `%include` and `%page`.

Unknown switch

Check command line for error.

UNTIL expected

Check repeat statement for proper form.

Use '.' after main program body

The indicated terminator is missing from end statement.

Use ';' to separate declarations

In addition to flagging the usual missing-semicolon errors, this message is issued when an illegal digit for the specified radix is found in the cross-hatch '#' format for constants.

Use ';' to separate statements

Value for qualifier out of range

A value supplied with the indicated command-line switch is not within the allowable range for that switch.

1914

Received of the Treasurer of the
Board of Directors of the
City of New York

the sum of \$100.00
for the purpose of

the purchase of

the sum of \$100.00

for the purpose of

the purchase of

the sum of \$100.00

for the purpose of

the purchase of

the sum of \$100.00

for the purpose of

the purchase of

the sum of \$100.00

for the purpose of

the purchase of

Appendix A: Compilation Error Messages

VAR parameters cannot be passed an expression or packed field

A var parameter must be the name of a variable or a component of a data structure. If the parameter is a component of a data structure (record or array), the structure may not be packed.

Variable name expected

Variable of type ARRAY expected

Variable of type RECORD expected

Variables of this type are not allowed in READ

Scalar variables may not be used in either read or write to a text file. Only predefined types (except boolean) and strings may be read from a text file.

Variables of this type are not allowed in WRITE

Scalar variables may not be used in either read or write to a text file. Only predefined types (except boolean) and strings may be read from a text file.

Variant label is undefined

Appendix B: Run-Time Error Messages

- 2 **Array subscript out of bounds**
An array index is outside of the limits established for the array in the **type** declaration that defines the array.

- 44 **Attempt to access block > 65535**
RT-11 files cannot contain more than 65535 blocks.

- 17 **Attempt to read past end of file**
An input operation was attempted on a file when **eof** is true. This is usually due to a logic error in the program and can often be solved installing checks for **eof**. This error can be trapped with the **noioerror** procedure.

- 15 **Attempt to write past end of file**
Files cannot be dynamically expanded. To increase the number of blocks allocated to the file, specify a larger value for the fourth parameter on the **rewrite** statement that opens the file.

- 33 **Attempted reference through NIL pointer**
A pointer variable was improperly used while its value was undefined or **nil**. This error could be the result of a pointer being disposed of before it is used, or of a value never being assigned to it. This could also occur if the pointer was created with **loophole** or **ref**. The **\$nopointercheck** switch suppresses this error message.

- 41 **Can't delete file**
The specified file cannot be deleted. This error can be trapped with the **noioerror** procedure.

- 11 **Can't open file**
The file could not be opened for the reason identified by the I/O error code. For input files, this error usually occurs if the file does not exist. You can trap this error by specifying and checking the fourth parameter on the **reset** or **rewrite** statement used to open the file.

- 42 **Can't rename file**
The file could not be renamed for the reason given by the I/O error code. This error can be trapped with the **noioerror** procedure.

- 37 **CASE selector matches no label**
A **case** selector expression has no matching **case** label. The **otherwise** clause can be used to detect this error. The **\$norangecheck** switch disables the detection of this error.

- 30 **Compiler/library mismatch**
The compiler version used to compile the main program does not match the support library used when the program was linked. This error could occur if a new version of Pascal-2 is installed on your system, and you attempt to link a program with a module compiled with an older version of the compiler. The solution here is to recompile all of your modules. This error could also happen if the compiler or library were updated independently. You might have to rebuild the compiler for your system.

- 35 **DISPOSE() of a NIL pointer**
The pointer value does not point into the heap memory pool. This error could occur if the pointer is not initialized (via **new**) or if the pointer was created with **loophole** or **ref**.

Received of the
Library of the
New York Public Library
the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

This book was purchased
for the Library of the
New York Public Library
by the sum of \$10.00
for the purchase of
the book entitled
"The History of the
United States of America
from 1776 to 1876"
by John P. Kennedy

Appendix B: Run-Time Error Messages

5 Division by zero

Division by zero is not defined.

22 Double deallocation of dynamic memory

The pointer variable points to an area of memory already available for reuse. Possibly the pointer was not initialized, or it was created with `loophole` or `ref`. Also, the heap may have been corrupted. This can happen if you make assignments using uninitialized pointers or if an `external` procedure is called and the number of parameters passed to the procedure differs from the procedure definition.

18 Error reading file

An I/O error was detected while your program was reading an input file. The I/O error code describes the exact cause of the error. This error is most often reported during a `read` or a `get` operation. This error can be trapped with the `noioerror` procedure.

19 Error writing file

An I/O error was detected while your program was writing an output file. The I/O error code describes the exact cause of the error. This error is most often reported during a `write` or `put` operation. This error can be trapped with the `noioerror` procedure.

8 EXP() overflow

The parameter passed to the `exp` routine would cause an overflow condition during the calculation of the `exp`. The maximum value permitted is approximately 88.

-- Fatal initialization error

This error indicates that the Pascal support library could not properly initialize the program for the reason given. This error usually occurs when the program is too large.

27 File is not a random access file. Use /SEEK

This error is caused by an attempt to use the `seek` procedure on a file that was not opened with the `/seek` I/O control switch. The `/seek` switch must be used with the `reset` or `rewrite` statement that opened the file. This error can also occur if you attempt to open a sequential device such as a terminal or printer using `/seek`.

38 File is not an input file

An input operation was attempted on a file that has not been prepared for reading by `reset`. Be sure to use the `/seek` I/O control switch when you are opening random access files.

39 File is not an output file

An output operation was attempted on a file that has not been prepared for writing by `rewrite`.

14 File name syntax error

The file name is not a valid file specification. Check the file specification for invalid characters or other garbage in the file name. You can trap this error by specifying the fourth parameter on the `reset` or `rewrite` statement used to open the file.

29 File not open

All files other than input and output must be opened with `reset` or `rewrite` before they can be accessed.

Handwritten text at the top of the page, possibly a title or header.

First main paragraph of handwritten text, starting with a capital letter.

Second main paragraph of handwritten text, starting with a capital letter.

Third main paragraph of handwritten text, starting with a capital letter.

Fourth main paragraph of handwritten text, starting with a capital letter.

Fifth main paragraph of handwritten text, starting with a capital letter.

Sixth main paragraph of handwritten text, starting with a capital letter.

Seventh main paragraph of handwritten text, starting with a capital letter.

Eighth main paragraph of handwritten text, starting with a capital letter.

Ninth main paragraph of handwritten text, starting with a capital letter.

Pascal-2 V2.1/RT-11 Programmer's Guide

- 6 **Floating point format error**
The program attempted to read a real number from a text file, where the data in the file is not a valid real number. This error can be trapped with the `noioerror` procedure.
- 3 **Floating point overflow**
The result of a floating-point operation is too large to represent as a real number. The magnitude of the largest real number is approximately $1.7E + 38$.
- 25 **Floating point support error**
This error results from a floating-point error condition other than real overflow, integer overflow or division by zero.
- 32 **I/O transfer error**
A hard error (i.e., device not ready, timing error, hardware read or write error) has occurred on the channel doing the transfer. Consult the *RT-11 Software Support Manual* for further assistance.
- 23 **Illegal value for integer**
The program attempted to read an integer value that lies outside the range $-32767..32767$. This error can be trapped with the `noioerror` procedure.
- 9 **LOG() of zero or a negative number**
Logarithms are only defined for positive values.
- **Multiple errors detected. Program aborted.**
This error occurs when an error is detected while another error is being processed. Rather than printing a possibly infinite list of errors, the support library prints this special error message and terminates the program. This error can be caused when the support library code has been accidentally overwritten.
- 21 **NEW() of zero length**
This error usually indicates an internal error in the Pascal support library. It could also be caused by an incorrect call to the `p$inew` function.
- 1 **Not enough memory.**
The `new` procedure was unable to allocate the requested block of memory. Dispose of noncritical memory or decrease the number of open files.
- 43 **Odd address or nonexistent memory trap**
An invalid memory location has been referenced. This error is the same as the PDP-11 "trap to 4" error.
- 40 **RENAME/DELETE of non-disk file**
The use of `rename` and `delete` is restricted to random-access devices only. Here the program attempted to perform a illegal (and meaningless) `rename` or `delete` operation on a non-disk device such as a line printer.
- 28 **Reserved instruction execution**
Several problems could cause this error. Check for the improper use of overlays or a mismatch between external procedure definitions and references. If this error happens on a statement involving real numbers, you may have configured your Pascal-2 system incorrectly.

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst.

and in reply to inform you that the same has been forwarded to the proper authorities for their consideration.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

[Faint text, possibly a date or reference]

[Faint text, possibly a closing or signature]

[Faint text, possibly a date or reference]

[Faint text, possibly a closing or signature]

[Faint text, possibly a date or reference]

[Faint text, possibly a closing or signature]

[Faint text, possibly a date or reference]

[Faint text, possibly a closing or signature]

[Faint text, possibly a date or reference]

Appendix B: Run-Time Error Messages

- 26 **SEEK() out of range**
The program attempted to "seek" a nonexistent record. Record numbers begin with 1 and end with the last record of the file. Attempts to access record numbers less than 1 or greater than the last record (past the end of file) will cause this error.
- 24 **Set element out of range**
The program attempted to reference an element of a **set** that is outside the range of values permitted in the set. The valid range is 0..255.
- 7 **SQRT() of a negative number**
The square root of a negative number is undefined.
- 37 **Stack overflow**
This error could be caused by an overly large program, excessive use of dynamic memory, or deeply nested recursion. If appropriate, try overlaying the program, or close unused files and dispose of unused memory.
- 16 **Too many files open**
No more channels are available to the program. Sixteen channels are normally available (0 through 15) unless the program is overlaid, in which case the operating system uses channel 15 for overlays. Close any unused files.
- 20 **TRUNC/ROUND overflow**
The result of a **trunc** or **round** operation is too large to be represented. Only real numbers in the range -32767.0 to 32768.0 may be converted to integers with the **trunc** or **round** functions.
- **Unknown Pascal run-time error #num**
This message indicates that the detected error has no corresponding error message text. This indicates an internal error in the support library. Contact Oregon Software or file a Trouble Report.
- 10 **Unrecognized file switch**
An I/O control switch specified on a **reset** or **rewrite** statement is unknown to the file system. Check the spelling of your file switches. You can trap this error by specifying the fourth parameter on the **reset** or **rewrite** statement that opened the file.
- 34 **Variable subrange exceeded**
The program attempted to assign a value to a variable that is outside the bounds of the subrange type. This error is often caused by uninitialized variables or the improper use of subrange definitions. The **\$norangecheck** switch disables the detection of this error.

Handwritten header or title at the top of the page.

Main body of handwritten text, consisting of several paragraphs. The text is mostly illegible due to fading and blurring.



Appendix C: Compiler Errors

Overflow Errors

Very complex or very large programs may exceed the capacity of the Pascal-2 compiler. Overflow of this sort is reported directly to the terminal rather than to the listing or error file. The compiler reports the type of overflow along with the name of the procedure causing the problem. Overflow errors may also occur in the main program. The following list of error messages assumes that a procedure named `MuchTooComplicated` has caused an overflow:

- Too many keys in procedure `MuchTooComplicated`
- Out of memory in procedure `MuchTooComplicated`
- Too many labels in procedure `MuchTooComplicated`
- Too many nodes in procedure `MuchTooComplicated`
- Code too complex in procedure `MuchTooComplicated`
- Too much object code in procedure `MuchTooComplicated`
- Too many Pascal labels in procedure `MuchTooComplicated`
- Too many external references in procedure `MuchTooComplicated`

An overflow condition in the main program will be reported as:

- Too many keys in main program

If compilation of a program causes one of the above error conditions, simplify the offending procedure or main program section. Two suggested ways to do this are to split the routine into several sub-procedures and/or reduce the number of type definitions.

Consistency Checks

In addition to the above error messages, consistency checks within the compiler can (in theory) trigger one of these errors:

- Undeleted temps in main program
- Internal temp error in main program
- Travrs build error in main program
- Travrs walk error in main program
- Bad adjust offset value in main program
- nnn consistency checks detected

You should seldom, if ever, see consistency-check errors; they are documented here for the sake of completeness. If you do see such an error, please send us a Trouble Report immediately. Along with the Trouble Report, send us the smallest possible source program that reproduces the error. Programs longer than one page should be sent on floppy disk or magnetic tape. (You also may call Oregon Software at 503-228-7760, but we will undoubtedly need to have the problem in writing.)

Appendix D: Default File Extensions

Appendix D: Default File Extensions

The default file extensions listed here apply to files generated by and/or referenced by Pascal-2 and its utilities. The first column lists the type of data contained in the file. The second column lists the extension.

<u>File</u>	<u>Extension</u>
Document	.DOC
Executable	.SAV
Listing	.LST
MACRO-11 Source Code	.MAC
Object	.OBJ
PROCREF Output	.PRF
Profiler Output	.PRO
PROSE Input	.PRS
Source	.PAS
Symbol Map	.SMP
Symbol Table	.SYM
Temporary	.TMP
XREF Output	.CRF

January 1900

1. The first of the year

2. The second of the year

3. The third of the year

4. The fourth of the year

5. The fifth of the year

6. The sixth of the year

7. The seventh of the year

8. The eighth of the year

9. The ninth of the year

10. The tenth of the year

11. The eleventh of the year

12. The twelfth of the year

Appendix E: Entry Points in the Pascal Support Library

<u>Entry Point</u>	<u>Description</u>
P\$0	Read character from standard file input
P\$1	Double-precision division simulation
P\$2	Read character from file
P\$3	Double-precision multiplication simulation
P\$4	Read integer from standard file input
P\$5	Double-precision subtraction simulation
P\$6	Read integer from text file
P\$7	Double-precision real addition simulation
P\$8	Read real number from standard file input
P\$9	Read double-precision real from standard file input
P\$10	Read real number from text file
P\$11	Read double-precision real from text file
P\$12	Read string from standard file input
P\$13	Permanently undefined (unlucky)
P\$14	Read string from text file
P\$15	Reserved
P\$16	Readln on standard file input
P\$17	Reserved
P\$18	Readln from text file
P\$19	Reserved
P\$20	Write character to standard file output
P\$21	Reserved
P\$22	Write character to text file
P\$23	Reserved
P\$24	Write integer to standard file output
P\$25	Reserved
P\$26	Write integer to text file
P\$27	Reserved
P\$28	Write real number to standard file output
P\$29	Write double-precision real to standard file output
P\$30	Write real number to text file
P\$31	Write double-precision real to text file
P\$32	Write string to standard file output
P\$33	Initialize standard files input and output
P\$34	Write string to text file
P\$35	Set user-handling of I/O errors for file (noioerror)
P\$36	Writeln to standard file output
P\$37	Status check of last file operation (ioerror)
P\$38	Writeln to text file
P\$39	I/O error code of last file operation (iostatus)
P\$40	Reserved
P\$41	"Stack overflow" error message
P\$42	Reserved
P\$43	"Subscript out of bounds" error message
P\$44	Reserved

Appendix E: Entry Points in the Pascal Support Library

<u>Entry Point</u>	<u>Description</u>
P\$45	"Variable subrange exceeded" error message
P\$46	Reserved
P\$47	"Reference through a nil pointer" error message
P\$48	Reserved
P\$49	"Case selector" error message
P\$50	Reserved
P\$51	"Division by zero" error message
P\$52	Reserved
P\$53	Delete file
P\$54	Reserved
P\$55	Rename file
P\$56	Reserved
P\$57	Close files in specified range
P\$58	Reserved
P\$59	Initialize Pascal
P\$60	Put next record
P\$61	Get next record
P\$62	Break file
P\$63	Program termination
P\$64	Rewrite file
P\$65	Seek record in file
P\$66	Reset file
P\$67	Debugger initialization
P\$68	Close file
P\$69	Debugger procedure entry
P\$70	New memory allocation
P\$71	Debugger procedure exit
P\$72	Dispose memory deallocation
P\$73	Debugger non-local goto
P\$74	Reserved
P\$75	Save registers
P\$76	Reserved
P\$77	Restore registers
P\$78	Signed integer multiply
P\$79	Reserved
P\$80	Signed integer divide
P\$81	Pack
P\$82	Signed integer mod
P\$83	Unpack
P\$84	Floating compare simulation
P\$85	Double-precision floating compare simulation
P\$86	Trunc of real number
P\$87	Trunc of double-precision real
P\$88	Float conversion to real
P\$89	Float conversion to double-precision real
P\$90	Sqrt of real number
P\$91	Sqrt of double-precision real
P\$92	Sin of real number
P\$93	Sin of double-precision real
P\$94	Cos of real number

Pascal-2 V2.1/RT-11 Programmer's Guide

<u>Entry Point</u>	<u>Description</u>
P\$95	Cos of double-precision real
P\$96	Atn (arctangent) of real number
P\$97	Atn of double-precision real
P\$98	Exp (exponential) of real number
P\$99	Exp of double-precision real
P\$100	Reserved
P\$101	Reserved
P\$102	Ln (natural logarithm) of real number
P\$103	Ln of double-precision real
P\$104	Reserved
P\$105	Reserved
P\$106	Time function - real
P\$107	Time function - double-precision real
P\$108	Round of real number
P\$109	Round of double-precision real
P\$110	Write boolean to standard file output
P\$111	Fortran interface
P\$112	Write boolean to text file
P\$113	Error reporting
P\$114	Reserved
P\$115	Reserved
P\$116	Unsigned integer multiplication simulation
P\$117	Real division simulation
P\$118	Unsigned integer division simulation
P\$119	Real multiplication simulation
P\$120	Unsigned integer mod
P\$121	Real subtraction simulation
P\$122	Reserved
P\$123	Real addition simulation
P\$124	Reserved
P\$125	Reserved
P\$126	Reserved
P\$127	Check for stack overflow
P\$128	Reserved
P\$129	Reserved
P\$130	Reserved
P\$131	Reserved
P\$132	Reserved
P\$133	Reserved
P\$134	Reserved
P\$135	Reserved

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

Pascal-2 V2.1/RT-11 Language Specification

Introduction to the Language Specification

The Pascal-2 compiler processes the standard Pascal language, as described in the *Pascal User Manual and Report* [2nd edition], by Kathleen Jensen and Niklaus Wirth, published by Springer-Verlag, corrected printing of 1978. This language is more completely described in ISO Draft Proposal 7185, ISO/TC 97/SC 5, dated August 12, 1982, which we call the "draft standard" hereafter.

Compliance is Level 1: conformant array parameters are included. Pascal-2 includes the extensions detailed in this guide. This guide includes data on non-standard language features. This guide is not intended as a full language document.

Syntax definitions in this specification use the notation described in Appendix C, Pascal-2 Syntax.

Changes in the Standard

Because you may not be familiar with all the changes to the Pascal language from Jensen and Wirth (1978) to the most recent draft of the standard (1982), this section outlines those changes and Pascal-2's method of implementing them.

'For' Statement Control Variables

Variables that control a **for** statement must be simple variables, local to the routine in which the **for** statement is written. Originally, any variable could be used.

File Declaration

The standard states that the files **input** and **output** are automatically declared as global variables if they are mentioned in the program heading. Because program headings are optional in Pascal-2, **input** and **output** are declared as global variables in every Pascal-2 program. Thus, you cannot redefine **input** or **output** at the global level. In earlier versions of the language, the actual point of definition was undefined.

Parameter Compatibility

The compatibility rules for **var** parameters are now defined according to a restrictive rule, which requires the argument passed to have the same type as the formal parameter. Although the types must be the same, the type identifiers may differ. The appearance of a new type construct creates a new type. Previously, the rules for **var** parameters were undefined.

1. The first part of the paper is devoted to a general discussion of the problem of the origin of life. It is shown that the problem is one of the most important and interesting in the history of science.

2. The second part of the paper is devoted to a detailed discussion of the various theories of the origin of life. It is shown that the most plausible theory is that of the spontaneous generation of life from non-living matter.

3. The third part of the paper is devoted to a discussion of the evidence in favor of the spontaneous generation of life. It is shown that the evidence is very strong and that the spontaneous generation of life is a fact.

4. The fourth part of the paper is devoted to a discussion of the various objections to the spontaneous generation of life. It is shown that the objections are not valid and that the spontaneous generation of life is a fact.

5. The fifth part of the paper is devoted to a discussion of the various theories of the origin of life. It is shown that the most plausible theory is that of the spontaneous generation of life from non-living matter.

Procedure and Function Parameters

The draft standard has changed the method of declaring procedure and function parameters. The new syntax provides a way of checking the parameters of these procedures and functions, thus reducing the likelihood of type errors.

The syntax for a parameter list is changed to:

```
parameter-list = "(" parameter-section {"," parameter-section } ")" .  
parameter-section = ( [ "var" ] identifier {"," identifier } ":" ( identifier  
| conformant-array-schema ) ) | procedure-heading | function-heading .
```

A full procedure heading must be provided for any procedure or function declared as a parameter, and the procedure heading for any procedure or function passed as an actual parameter must match. For example:

```
var  
  K, L: integer;  
  
  procedure P(procedure Q(I, J:integer));  
  begin  
    Q(K, L);  
  end;  
  
  procedure P1(I,J: integer);  
  begin  
    writeln('test of proc parameters', I, J);  
  end;  
  
begin  
  K := 1;  
  L := 20;  
  P(P1);  
end.
```

The program issues the following output:

```
test of proc parameters      1      20
```

The draft standard does not allow a standard function to be used as a parameter for a function or procedure. To pass a standard function as a function or procedure argument, you must define a function that calls the standard function, then pass the user-defined function as the function or procedure argument.

Conformant Array Parameters

Normally, a procedure or function accepts an array parameter containing a fixed number of elements. The number of elements holding meaningful information may vary but the size of the array may not. If you need to pass arrays of different lengths, you have to declare and pass a general array that is as long as the longest possible array, and you must track the last element of each. Another approach is to write a separate procedure to handle each size of array, which is clearly inefficient.

Use of conformant array parameters solves this problem. Conformant array parameters are formal parameters that allow you to write a general procedure or function that, at each activation, accepts array parameters of different size and with different lower and upper bounds. At activation, the upper and lower bounds of the conformant array parameter assume the upper and lower bounds of the passed parameter (the actual parameter).

Handwritten header or title at the top of the page.

First main paragraph of handwritten text, starting with a capital letter.

Second main paragraph of handwritten text, continuing the narrative or argument.

Third main paragraph of handwritten text, concluding the page.

The syntax for a conformant array parameter is:

```

conformant-array-parameter-specification =
    [ "var" ] identifier-list ":" conformant-array-schema .

conformant-array-schema = packed-conformant-array-schema
    | unpacked-conformant-array-schema .

packed-conformant-array-schema = "packed array"
    "[" index-type-specification "]" "of" type-identifier .

unpacked-conformant-array-schema = "array" "[" index-type-specification
    { ";" index-type-specification } "]" "of" ( type-identifier | conformant-array-schema
    ) .

index-type-specification = bound-identifier ".." bound-identifier ":" type-identifier .

```

As the EBNF diagrams show, a conformant array schema may be either packed or unpacked. An unpacked conformant array may be nested within itself or within other conformant arrays (either packed or unpacked); if so, an abbreviated form may be used. In the example below, **Mx** is the conformant array parameter being used in **Examp**. **T1**, **T2** and **T3** are data types. The two definitions are equivalent. Notice that the semicolon in the abbreviated form replaces '**]** **of** **array** [**]**' in the long form.

```

procedure Examp(var Mx: array [Lb1..Ub1: T1] of array [Lb2..Ub2: T2] of T3);
    or
procedure Examp(var Mx: array [Lb1..Ub1: T1; Lb2..Ub2: T2] of T3);

```

An array may be passed as a conformant array parameter if:

- the elements have the same types,
- the index types are compatible, and
- the bounds are within the range specified by the parameter declaration.

If two parameters are specified with a single conformant array schema, the actual parameter passed must have the same type. Also, a value conformant array may not be passed as a parameter to another procedure or function.

The next example demonstrates the use of conformant array parameters. The formal parameter **Arr** is a conformant array parameter and takes the values of two different-sized arrays, **First** and **Second**. At the first activation of the function **AddArray**, the two elements of array **First** are added together to reach a sum. The next activation adds up the four elements of array **Second** and arrives

The first part of the paper is devoted to a general discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The second part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The third part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The fourth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The fifth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The sixth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The seventh part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The eighth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The ninth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom. The tenth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of the atom.

Pascal-2 V2.1/RT-11 Language Specification

at a different sum, as shown in the output following the program listing.

```
program Conform;
var
  First: array [1..2] of integer;    { two-element array }
  Second: array [0..3] of integer;   { four-element array }
  Total: integer;

function AddArray(var Arr: array [Lower..Upper: integer] of integer): integer;
var
  I, Sum: integer;
begin
  Sum := 0;
  for I := Lower to Upper do
    Sum := Sum + Arr[I];
  AddArray := Sum
end;

begin
  First[1] := 5; First[2] := 9;
  Total := AddArray(First);           _____ called with two-element array
  writeln('Total for this array is: ', Total:5);
  Second[0] := 1; Second[1] := -31; Second[2] := 77; Second[3] := 15;
  Total := AddArray(Second);          _____ called with four-element array
  writeln('Total for this array is: ', Total:5)
end.
```

Running the program yields:

```
Total for this array is:    14  _____ sum of elements of array First
Total for this array is:    62  _____ sum of elements of array Second
```

For a practical example of the use of conformant array parameters, see the source code of Pascal-2's Dynamic String Package, in the file STRING.PAS.

Literal Strings

A literal string may not extend over more than a single line. Earlier standards were unclear on this point. The limitation allows better diagnostics for unterminated strings.

'Write,' 'Writeln' of 'Packed Array of Char'

A `write` or `writeln` procedure call applied to a `packed array of char` writes only as many characters as the field-width parameter specifies. If the `packed array of char` exceeds the field-width, the string is truncated. The string is right-justified if the specified field width is longer than the packed array. If no field width is specified, a `write` or `writeln` writes as many characters as are in the string.

Handwritten header or title at the top of the page.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Fifth main paragraph of handwritten text.

Sixth main paragraph of handwritten text.

Example:

```

program Buff;
var
  Buffer: packed array [1..30] of char;
  BuffCount: integer;

begin
  Buffer:= 'This is a packed array of char';
  writeln(Buffer);
  BuffCount := 6;
  writeln(Buffer:BuffCount);
  writeln('cutoff':3);
  write('shorter':10);
end.

```

When executed, the program yields these results:

```

This is a packed array of char
This i
cut
shorter _____ note leading blanks

```

Identifiers

The initial character of an identifier must be an alphabetic character or a dollar sign. All other characters making up identifiers may be any combination of digits, letters, dollar signs or underbars. Identifiers may be of any length; all characters are significant. Lower-case characters are interpreted in the same way as upper-case characters. For example, `name`, `Name`, `NaME`, and `NAME` are equivalent. See "Syntax Extensions" for details on the use of the non-standard dollar sign and underbar in identifiers.

Alternate Symbol Representations

The standard now defines alternate representations for symbols that are unavailable in some character sets. These are:

<u>Standard Symbol</u>	<u>Alternate Symbol</u>
~ or †	@ ('at' sign)
{	(*
}	*)
[	(.
]	.)

The alternate comment delimiters are equivalent to the standard comment delimiters, and a comment may open with one type of delimiter and close with the other. Comments may not be nested.

Examples:

```

(* This is a valid comment }
{ This is (* not *) a valid comment }

```

Handwritten text at the top left of the page.

Handwritten text in the upper middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Handwritten text in the lower middle section.

Implementation Definitions

This section provides details and characteristics of implementation-defined elements of Pascal-2.

Standard Type 'Integer'

The predefined identifier `maxint` has the value 32767.

The standard type `integer` has the range (-32767..32767). An unsigned (extended-range) integer may be defined with the range 0..65535. See "Unsigned Integer Conversion" in the Programmer's Guide.

Standard Type 'Real'

A `real` variable has the standard PDP-11 single-precision or double-precision floating-point structure, with magnitude in the range $1\text{E}-38..1\text{E}+38$. Single-precision values give approximately 7 decimal digit precision; extended (double-precision) values give approximately 15-digit precision. Arithmetic overflow is detected for all `real` operations, but underflow is ignored and returns a result of zero.

The standard transcendental routines are accurate to 6 decimal digits in single precision and to 15 decimal digits in extended precision.

Standard Type 'Char'

The draft standard does not define the character set to be used internally to represent `char`. Pascal-2 uses 8-bit characters, allowing the use of the extended version of the ASCII character set, rather than 7-bit characters to represent the standard ASCII character set. The most significant bit is "off" unless used with extended character sets. `Ord(char)` is in the range 0..255.

Programs that calculate bit or byte offsets into a packed structure should treat a character as 8 bits, not 7; and storage size is the same for characters in either packed or unpacked structures.

Standard Type 'Text'

The standard type `text` is a file type with components of type `char`. `Text` is implemented as a file of 8-bit ASCII characters.

'Set' Types

Pascal-2 limits a `set` to a maximum of 256 elements. The lower and upper bounds must lie in the range 0..255, e.g., `set of 4..9`. The declaration `set of integer` is equivalent to the declaration `set of 0..255`. See "Undetected Errors" for restrictions on the checking of integer sets.

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

THE HISTORY OF ARTS AND ARCHITECTURE

I/O Definitions

The following table summarizes the default field widths used when values are written to a text file:

<u>Value Type</u>	<u>Field Width</u>
integer	7
real	13
boolean	5

The floating-point representation of a real number includes the sign of the number (a space for positive numbers and a '-' for negative numbers), the real number in scientific notation, an upper-case E signifying exponential notation, the sign of the exponent ('+' or '-'), and a two-digit exponent. For example, the real number -105.39 prints as -1.053900E+02.

Boolean values are written in upper case (TRUE, FALSE). In the five-character default field, the value TRUE is right-justified, with a leading blank before the 'T'.

The procedure **page(F)** inserts a form feed (page eject) into the file specified by the required file argument. Calling **page(F)** with data in the file buffer executes **writeln(F)**, which writes the remainder of the buffer, and **write(F,chr(12))**, which writes the form-feed character. Calling **page(F)** with an empty file buffer results in a page eject only.

If associated with the standard input file (the terminal), **reset(input)** performs the equivalent of a **readln**, but otherwise has no effect; in the same way, **rewrite(output)** prints any incomplete line, but otherwise has no effect. **Reset(output)** or **rewrite(input)** produces an error message. See "External File Access" for details on the use of the extended form of **reset(input)** or **rewrite(output)**.

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

1870-1871

Pascal-2 V2.1/RT-11 Language Specification

Syntax Extensions

This section describes Pascal-2 extensions to the syntax of standard Pascal.

Identifiers

The character \$ (dollar sign) is allowed in an identifier anywhere an alphabetic character is allowed. The character _ (underbar) is allowed anywhere a numeric character is allowed. For example, the identifier `_ABC` is not valid because it begins with an underbar. The following are legal identifiers:

```
system$name
$$file
this_is_a_long_identifier
This___Is_Also___Legal
```

Program Heading

In standard Pascal, the program heading is required, and the parameters define the external files to be used:

```
program Test (input, output, File3);
```

In Pascal-2, the program heading and parameters are not required. If present, they will be checked for proper syntax. The file parameters will otherwise be ignored. `Input` and `output` are automatically declared file variables. Every other external file must be specified by an additional parameter allowed in the standard procedures `reset` and `rewrite`. See "External File Access" under "I/O Support Extensions" for details.

Though not required, inclusion of the program name on the program statement is still a good practice because it names the object module for main programs and external modules. Further, the program name is used to name the psect when the `own` compilation switch is specified.

Declaration Order

The declaration sections `label`, `const`, `type`, `var`, `procedure`, and `function` may be interleaved as desired at the global level of a program. `Const` and `type` may be interleaved at other levels. This extension is useful for source module inclusion and structured constant definitions as described below. Any number of declaration sections of each type may be present. An identifier still must be defined before the identifier is used in any other way.

'%Include' Lexical Directive

A special directive allows separate text files to be included within a program. The contents of the separate file are inserted into the program at whatever point the `%include` directive occurs. Included files may themselves contain `%include` directives, nested to a maximum of seven levels.

The syntax for the `%include` directive is:

```
%include 'file-name-string';
```

The *file-name-string* must contain at least the name of the file; if no file extension is specified, `.PAS` is assumed. In addition to the file name and extension, *file-name-string* may contain such information as the logical device name and disk volume number of the file.

The single quotes ('...') enclosing *file-name-string* are optional. This syntax provides compatibility with other implementations of Pascal-2 that allow file version numbers.

Examples:

```
%include hdr;
%include 'makhdr.pas';
%include tern.doc;
%include 'sy:hfil';
```

See the Programmer's Guide for details.

'%Page' Lexical Directive

The `%page` directive causes a page break (form feed) in the listing file immediately following the line on which the `%page` directive is placed. The `%page` directive itself is printed in the listing file on the last listed line of the page preceding the page eject. The ending semicolon is optional.

'External' and 'NonPascal' Directives

Similar to the `forward` standard directive, the `external` directive distinguishes a particular Pascal procedure or function that is separate from the module that invokes it. An external procedure must be declared at the global level. If the body of an external procedure or function does not appear in a compilation, it is assumed that the body will be in another object module. If the body of the external procedure does appear, its name will be made available in the object module for reference by other modules. References to the external procedure are resolved at link time.

Limitations of the object module structure require that external names be distinct within the first six characters. The underbar cannot be expressed in the object module format and is replaced by a period in the external name. No type checking is done for parameters of an external routine.

The `nonpascal` directive is used instead of `external` if the external procedure is written in a language other than Pascal. `Nonpascal` creates an interface between the Pascal-2 calling sequence of the program or module doing the calling and the DEC calling sequence required by non-Pascal external routine, usually a FORTRAN or MACRO-11 module. The `nonpascal` directive makes use of the convention of having register R5 point to a list of parameters. All parameters are passed by reference, so only `var` parameters may be used. MACRO-11 routines written with the Pascal-2 PASM utility must be declared as `external` rather than `nonpascal`, because PASM simulates the Pascal-2 calling sequence.

See also "External Modules" in the Programmer's Guide for details.

Structured Constants

The syntax for constant definitions is extended to allow you to specify constants in record and array types. Under the standard, arrays or records cannot be assigned values in the constants declarations; each element must be assigned a value in the program body with an assignment statement. The structured-constants language extension eliminates the need to use assignment statements to assign values to constants of type array or record. See the examples following for a comparison of structured constants declarations versus standard constant declarations.

100-100000

100-100000

100-100000
100-100000
100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

100-100000

Pascal-3 V3.1/RT-11 Language Specification

The formal syntax for structured constants is:

```
structured-constant =  
    structured-type-identifier constant-component-list .  
constant-component-list = "(" constant-component { "," constant-component } ")" .  
constant-component = constant | constant-component-list .
```

where

structured-type-identifier

Is a data type with an array or record structure. All of the components of that structure must be of simple types, array types, or record types.

constant-component

Must correspond one to one with the component of the structured (array or record) type, and each *constant-component* must be a constant of the same type as the corresponding structure component. An access to the structure component returns the value of the *constant-component*. If the structure component is of a structured type, only the corresponding *constant-component-list* must be provided, declared with the proper syntax.

The following are valid declarations. Note that the data types needed by the structured constant must be declared before the structured constants.

```
type  
    S1 = packed array [1..4] of char;  
    S2 = record  
        String: S1;  
    end;
```

```
const  
    C1 = S1('a', 'b', 'c', 'd');  
    C2 = S2('abcd');
```

The *structured-type-identifier* for individual components need not be provided. For variant records (even those without a tag-field) a tag value must be provided in the *constant-component-list*.

Constants used as components in a *constant-component-list* appear between nested levels of parentheses. If an element is another structured type, a constant type of the same structure may appear or its elements may be set individually between inner parentheses. However, you may not use structured constants or their individual elements as case labels; case labels must be of simple type.

Examples of Structured Constants

Three examples are presented showing several uses of structured constants. The first example illustrates the nesting of parentheses in the structured constant declarations. The second example compares the standard's method of declaring record or array constants with the structured constant method. The third example shows the correct way to declare multidimensional arrays of constants.

The structured constant *Workers* in the following declarations contains three levels of parentheses: the first for the array structure; the next for the outer record; the last for the pay information. The fourth constant in the array, however, for *Maxine*, contains a structured element that is set by

Published by the American Medical Association, 535 North Dearborn Street, Chicago 10, Ill.

Second-Class Postage Paid at Chicago, Ill., and at additional mailing offices. Postmaster: Send address changes in this journal to JOURNAL OF THE AMERICAN MEDICAL ASSOCIATION, 535 North Dearborn Street, Chicago 10, Ill.

Acceptance for mailing at special rate of postage provided for in Section 1103, Act of October 3, 1917, authorized on July 11, 1956.

Postage paid by addressee

Chicago, Ill.

Postage paid by addressee

Chicago, Ill.

Copyright © 1957 by American Medical Association

Reproduction of this journal in whole or in part without the written permission of the American Medical Association is prohibited.

Printed in the United States of America

The Journal of the American Medical Association is a weekly publication of the American Medical Association. It is published for the Association by the American Medical Association, 535 North Dearborn Street, Chicago 10, Ill. The Journal is a national publication and is distributed to all members of the American Medical Association. It contains a wide variety of material, including original articles, reviews, and news items. The Journal is a valuable source of information for all physicians and is an essential part of the medical profession's literature.

reference to a constant of the same type with no further inner parentheses.

```

type
  Compensation = (Paid, Unpaid);
  Paytype = Record
    Title : (Clerk, Indian, Chief, President);
    case Compensation of
      Paid: (Rate: real);
      Unpaid: ();
    end;
  Employeetable = array[1..4] of record
    Name : packed array[1..10] of char;
    Payinfo : Paytype;
  end;

const
  Conchief = Paytype(Chief, Paid, 6.85);  — note redefinition for Maxine
  Workers = Employeetable(
    ('Charlie', (Clerk, Paid, 3.40)),
    ('Samuel', (Indian, Paid, 5.25)),
    ('Edward', (President, Unpaid)),
    ('Maxine', Conchief) ————— note condensed form
  );

```

To illustrate the efficiency and ease of use of structured constants, we present a comparison of the standard method of declaring constants for arrays and records versus the structured constants method. The program used in the comparison — **DayCalc** — calculates the day of the week for any date. The declarations below are those required to declare two arrays of constants, **MonthName** and **DayOffset**. Note that the type declarations are identical in both cases. The declaration of other data types, constants and variables have been omitted.

To conform to the standard, constants in arrays and records must be declared and assigned values as shown below. This method requires many more statements than the equivalent structured constants declarations, provided following the standard example.

```

program DayCalc;      { use of standard constant declarations }
type
  Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
           Unknown);
  Name = packed array [1..3] of char;
  NameList = array [Month] of Name;
  DayOffsetList = packed array [Month] of 0..6;

var
  MonthName: NameList;      { Text for name of month }
  DayOffset: DayOffsetList; { Day mod 7 }

```

1890

Received of the Treasurer of the County of ...

the sum of ...

for ...

...

...

...

...

...

...

...

...

...

...


```

begin    { DayCalc }

    MonthName[Jan] := 'jan'; MonthName[Feb] := 'feb'; MonthName[Mar] := 'mar';
    MonthName[Apr] := 'apr'; MonthName[May] := 'may'; MonthName[Jun] := 'jun';
    MonthName[Jul] := 'jul'; MonthName[Aug] := 'aug'; MonthName[Sep] := 'sep';
    MonthName[Oct] := 'oct'; MonthName[Nov] := 'nov'; MonthName[Dec] := 'dec';
    MonthName[Unknown] := '???';

    DayOffset[Jan] := 0; DayOffset[Feb] := 3; DayOffset[Mar] := 3;
    DayOffset[Apr] := 6; DayOffset[May] := 1; DayOffset[Jun] := 4;
    DayOffset[Jul] := 6; DayOffset[Aug] := 2; DayOffset[Sep] := 5;
    DayOffset[Oct] := 0; DayOffset[Nov] := 3; DayOffset[Dec] := 5;
    DayOffset[Unknown] := 0;

    : ----- rest of program goes here

```

With structured constants, on the other hand, your code is much shorter and easier to maintain than with the standard method. The only drawbacks are that the program is non-standard and is not necessarily portable to other Pascal implementations.

```

program DayCalc;    { use of structured constants }
type
    Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
             Unknown);
    DayOffsetList = packed array [Month] of 0..6;
    Name = packed array [1..3] of char;
    NameList = array [Month] of Name;

const
    MonthName = NameList('jan', 'feb', 'mar', 'apr', 'may', 'jun',
                        'jul', 'aug', 'sep', 'oct', 'nov', 'dec', '???');
    DayOffset = DayOffsetList(0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5, 0);

    : ----- rest of declarations and program body goes here

```

Multidimensional arrays of constants, such as the two-dimensional array below, can be declared as in the following program. This program prints the elements of the two-dimensional array `Table` in the order they are stored.

```

program TwoDimensions;
const
    MaxElem = 3;

type
    CharTable = packed array [1..MaxElem, 1..MaxElem] of char;

const
    Table = CharTable (('x', 'y', 'z'),
                      ('a', 'b', 'c'),
                      ('p', 'd', 'q'));

var
    I, J: integer;

```

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that this is crucial for ensuring the integrity of the financial system and for providing a clear audit trail.

In the second part, the document outlines the various methods used to collect and analyze data. It describes how these methods are designed to provide a comprehensive view of the system's performance and to identify any potential areas of concern.

The third part of the document focuses on the results of the analysis. It presents a series of charts and graphs that illustrate the trends and patterns observed in the data. These visual aids are intended to make the information more accessible and easier to understand.

Finally, the document concludes with a series of recommendations for future work. It suggests that further research be conducted to explore the underlying causes of the observed trends and to develop strategies to address any identified issues.

The document also includes a section on the limitations of the study. It acknowledges that there are certain constraints on the data and the analysis, and that these may affect the overall conclusions drawn from the study.

In addition, the document provides a brief overview of the background and context of the study. It explains the reasons for conducting the research and the specific objectives that were being pursued.

The document is organized into several sections, each of which covers a different aspect of the study. This structure is designed to allow readers to navigate the document easily and to find the information they are most interested in.

Overall, the document provides a detailed and thorough overview of the study. It covers all the key aspects of the research, from the initial data collection to the final conclusions and recommendations.

The document is written in a clear and concise style, using plain language to explain complex concepts. It is intended to be accessible to a wide range of readers, including those who are not experts in the field.


```

begin      { TwoDimensions }
  for I := 1 to MaxElem do
    for J := 1 to MaxElem do
      write(Table[I,J], ' ');
    writeln;
  end.      { TwoDimensions }

```

Running this program yields:

```

x y z a b c p d q

```

Default Case Label ('Otherwise')

A default statement can be included in a **case** statement according to the following syntax:

```

case-statement = "case" case-index "of" [ case-element { ":" case-element } ]
                 [ ":" ] [ "otherwise" default-statement [ ":" ] ] "end" .

```

The default statement, which immediately follows the **otherwise** clause, is executed if no case label matches the value of the *case-index*. A special note on **otherwise** syntax: In contrast to the case label, which requires a colon, the **otherwise** clause must not contain a colon or a compilation error results.

Example:

```

case I of
  1: Ch := ':';
  9: Ch := ':';
otherwise Ch := '*';
end;

```

The list *case-element* is optional, so the following example is valid.

```

case I of
otherwise I := 1;
end;

```


Pascal-2 V2.1/RT-11 Language Specification

I/O Support Extensions

I/O support extensions provide the Pascal-2 programmer with additional control of the interface to the operating system.

External File Access

The standard procedures **reset**, for opening an existing file, and **rewrite**, for creating a new file, have been extended to accept optional arguments that give Pascal-2 programs the ability to associate internal file variables with external file or device specifications. The syntax is as follows:

```
rewrite(file-variable, device-or-file-name, default-values, file-status);  
reset(file-variable, device-or-file-name, default-values, file-status);
```

where

file-variable

is a standard Pascal file variable.

device-or-file-name

specifies the name of an external file with which the file variable is to be associated. This parameter, which may be a device or file name specification, must be a string type and may be either a literal string or a variable.

default-values

is also of a string type, providing default values for any file fields not provided in the file name, including default file options.

file-status

is an integer variable that is primarily used to return a special status code if the file cannot be opened; this code allows a program to recover from an otherwise fatal error. The fourth parameter also may be used to determine the number of blocks allocated to a file or to specify the number of blocks to allocated to a new file. These uses are explained in detail below.

Commas must separate any optional parameters used; a comma must be included to mark off an omitted parameter but need not follow the last included parameter. The following example opens a file for direct access and skips the file-name parameter, indicating a temporary file.

```
rewrite(F1, , '/seek');
```

See "Random Access to Data Files" for details on the **/seek** switch, and see "I/O Control Switches" in the Programmer's Guide for details on the use of other file switches.

The optional parameters may be used to redirect the standard files **input** or **output**, which by default are file variables associated with the standard terminal input or output devices, respectively. The next example redirects output from the terminal to the line printer.

```
rewrite(output, 'LP:');
```

Normally, an I/O error with **reset** or **rewrite** causes the support library to trap the error, terminate the program, and print an error message and procedure walkback. The fourth parameter may be used to return control to the program. If the fourth parameter is specified and an I/O error occurs, the support library sets the value of the fourth parameter to -1 and returns control to the program. You must check the value returned by the fourth parameter and specify what action to take if an error occurs. For example:

```
reset(infile, Filename, '.lst', status);  — default extension of .LST  
if status = -1 then UserProcessError  — response needed to error status  
else ContinueUserProgram;
```


Or you may use the predefined functions in the support library to initiate run-time diagnostics. These functions check the status of the fourth parameter and respond accordingly. See "Run-Time Error Reporting" in the Programmer's Guide. Either way, you must check the value of the fourth parameter each time you use it; otherwise, the program continues but may act unpredictably.

If the file is successfully opened, the fourth parameter returns the number of blocks allocated to the file. In addition, the fourth parameter may be used with `rewrite` to specify the number of blocks to be initially allocated to the file. When the size of the file is known in advance, this specification allows efficient space allocation by the operating system. If the file does not actually occupy the number of blocks specified, however, the operating system will truncate the file to the number of blocks needed. In turn, the value of the fourth parameter may be checked after a `rewrite` to be certain that the file was allocated the number of blocks you wished.

Two values for the fourth parameter have special meaning on RT-11. A value of 0 indicates that the file should be allocated one-half of the largest contiguous space. A value of -1 specifies that you want all contiguous space allocated to the file. In both cases, the value upon return is the amount of space actually given to the file or is -1 if an error occurred in opening the file. If the fourth parameter is absent, the file size is determined by the operating system and expands dynamically. Examples:

```
reset(f, 'test', '.pas', size);  _____ assumes default of .PAS
writeln(size);  _____ returns the size of the file in blocks
:
size := 64;
rewrite(output, outstr, '.lis', size);  _____ file initially allocated 64 blocks
```

'Close' Procedure

The `close` predefined procedure indicates that its file parameter is no longer in use; `close` will reclaim buffer memory. Further access to the file is prohibited until `reset` or `rewrite` is used. Files are automatically closed upon program termination, or when they appear in another `reset` or `rewrite`; `close` allows files to be closed manually when it is necessary to reclaim buffer space before then. In addition, a file variable local to a procedure or function is automatically closed when that function or procedure terminates. See the sample program `Alphas` in the next section for implicit uses of `close`.

Random Access to Data Files ('Seek')

Pascal-2 includes the `seek` predefined procedure to allow direct access (random access) to data files opened with the `/seek` file control switch. The `seek` procedure requires two parameters: a file variable of the file to be accessed, declared as a file of `char` or other file type (but not of type `text`); and an integer record number (records in the file are numbered sequentially beginning with 1). After the `seek` call, the specified record is available in the file buffer if it exists; otherwise `eof` is set to indicate that the record is not available.

`Seek` also enables both reading and writing on the same file for in-place record updates. `Put` is required if the file buffer variable is to be written to the file. `Get` and `put` may be mixed with `seek` for sequential access, because the internal record pointer is updated after each `get` and `put`. See the example following for the use of `put` and `seek`.

After the file pointer is positioned by `seek`, both `read` and `write` as well as `get` and `put` may be performed. `Read` and `write` transfer data between the user variable and the file; `get` and `put` transfer data between the file buffer variable and the file. The following sequences may be used for direct access.

```
seek(F,I); read(F,V); { read record I into V }
seek(F,I); write(F,V); { write record I from V }
```


Pascal-2 V2.1/RT-11 Language Specification

Example:

```
program Alphas;
var
  C: char;
  F: file of char;

begin
  rewrite(F, 'alpha.txt');      { open F for writing }
  for C := 'a' to 'z' do
    write(F, C);                { write letters of the alphabet to F }
  reset(F, 'alpha.txt/seek');   { close and reopen F for seeking }
  seek(F, 4);                   { read record containing 'd' }
  writeln(F^);                  { write a 'd' to output }
  F^ := 'z';                    { 'd' becomes 'z' }
  put(F);                       { write 'z' to F in place of 'd' }
end.                            F closed automatically
```

As the program shows, the `/seek` file control switch must be used with `reset` or `rewrite` if the `seek` procedure is to be used to access the file. See "I/O Control Switches" in the Programmer's Guide for details.

At run-time, the character 'd' is written to the terminal. After program termination, the file ALPHA.TXT contains:

abczefghijklmnopqrstuvwxyz ————— z takes the place of d

`Seek` does not work on text files. For simulated random access on text files, you must use the `getpos` and `setpos` external procedures. See "Random Access to 'Text' Files" in the Programmer's Guide.

String Input ('Read' and 'Readln')

A character string is a packed array [1..n] of `char`. The `read` and `readln` procedures may be used to read variables of string types. Characters are read until the variable is filled. If `coln` becomes true, the remainder of the string is filled with spaces. See "The Dynamic String Package" in the Utilities Guide for more sophisticated ways to read and manipulate strings.

'Break' Procedure

For efficiency, Pascal-2 buffers transmitted output data. `Break(F)` forces the actual transmission of data from a partially filled buffer of file `F`. This can be useful with interactive terminals or to guarantee actual transmission of data to a shared disk file.

Octal Output

In an integer `write` procedure call, a negative field-width specification will represent characters in octal (base 8).

Example:

```
write(I:-5);           { Display octal value of I }
```


Real Number Formatting

If the second formatting field is negative, a real number is printed in scientific notation. The number of digits to the right of the decimal point is the number specified in the second field. (The standard allows you to specify an integer constant or an integer expression in either formatting field.)

For example,

```
write(R:20:-5);
```

prints R with one digit to the left of the decimal point and five digits to the right, followed by an upper-case E, a sign character '+' or '-' and two digits signifying the exponent. The entire number is right-justified in a 20-character field.

If R has the value -367.2, the statement `writeln('R=',R:20:-5)` prints:

```
R=      -3.67200E+02
```

1. The first part of the report deals with the general situation of the country and the progress of the work during the year.

2. The second part of the report deals with the results of the work during the year and the progress of the work during the year.

Low-Level Interface

This section describes Pascal-2 extensions that are useful to programmers needing access to machine-dependent characteristics.

Boolean Operators on Integer

The boolean operators `and`, `or`, and `not` may be applied to operands of `integer` or `integer` subrange type. The `not` operator is always applied first. The operators produce a 16-bit result of `integer` type.

Nondecimal Integer Constants

Nondecimal integer constants may be specified in two forms of notation. In the preferred form, the nondecimal value is written as shown:

nondecimal-integer-constant = *digit-sequence* "8" *hexadecimal-digit-sequence* .

where *digit-sequence* is the radix, or base, of the number, in the range 2..16. The number following the cross-hatch character '#' is any number represented in base *digit-sequence* notation. The '#' symbol is required regardless of base. For example, the decimal value 255 is written `8#377` for base 8 and `16#FF` for base 16. Also, the redundant form `10#255` is valid for the decimal value 255.

Pascal-2 supports another form of notation as a special case. Octal (base 8) notation for integer constants is signified by the suffix "B" (upper or lower case), so that `377B` and `377b` are the same value as 255 decimal.

Extended-Range Arithmetic

The normal range of `integer` variables in Pascal-2 is -32767..32767, but you also may declare `integer` types in the extended range of 0..65535. A variable with an upper limit greater than 32767 is called an extended-range or "unsigned" variable. Normal arithmetic operations, with the exception of division and modulo, are performed on extended-range variables. Comparisons and division are signed. An integer value may be assigned to an extended value, being converted as a bit pattern. If the value being assigned is negative, the error is not trapped at run-time, since there is no way for the compiler to tell the difference between a negative value and an extended-range value. The same sort of implicit transformation is true when an extended value is assigned to an integer variable. No conversion is done for constants.

The following sample program illustrates the way Pascal-2 handles extended-range numbers. Within the `repeat until` statement, the program reads an integer then prints it as an unsigned integer

and as a signed integer. The external procedure `Uwrite` is provided in the section on "Unsigned Integer Conversion" in the Programmer's Guide.

```

program BigNumberTest;
type
    Unsigned = 0..65535;

var
    BigNumber: Unsigned;

procedure Uwrite(X: Unsigned; Width: integer);
    external;    { procedure to write an unsigned integer to output }

begin { BigNumberTest }
    repeat
        write('Enter an integer: ');
        readln(BigNumber);
        write(' Unsigned, BIGNUMBER = ');
        uwrite(BigNumber,1); writeln;
        writeln(' Signed, BIGNUMBER  = ', BigNumber:1);
        writeln;
    until false { forever };
end.    { BigNumberTest }

```

The program is executed, producing the following results. As mentioned earlier, the allowable range of values for the integer `BigNumber` is `-32767..32767`. The final entry — in fact, any value outside the range of possible integers — is an invalid value for an integer, halting the program with a walkback (unless walkback is disabled).

```

Enter an integer: -1
Unsigned, BIGNUMBER = 65535
Signed, BIGNUMBER  = -1

```

```

Enter an integer: -32767
Unsigned, BIGNUMBER = 32769
Signed, BIGNUMBER  = -32767

```

```

Enter an integer: 32767
Unsigned, BIGNUMBER = 32767
Signed, BIGNUMBER  = 32767

```

```

Enter an integer: -5555
Unsigned, BIGNUMBER = 59981
Signed, BIGNUMBER  = -5555

```

```

Enter an integer: 5555
Unsigned, BIGNUMBER = 5555
Signed, BIGNUMBER  = 5555

```

```

Enter an integer: 65535

```

```

PASCAL--I/O error at user PC= 1050B
Illegal value for integer

```

```

Error occurred at line 16 in program bignumbertest

```


Pascal-3 V2.1/RT-11 Language Specification

See "Unsigned Integer Conversion" in the Programmer's Guide for Pascal routines that perform extended-range arithmetic and extended-range output.

"Origin" Declaration

A variable can be declared to have a particular address in the I/O page or system area with the following syntax:

var-declaration = *var-element* {*","* *var-element*}*":"* *type* .

var-element = *identifier* [*"origin"* *constant*] .

The constant in the above syntax must have an integer value. A variable so specified has the address given by the integer following *origin*. This must be in the system space 0..777B or in the I/O page 180000B..177777B.

The following example demonstrates the use of *origin*, plus the use of the *ref* and *size* functions. See "Ref Function" and "Size and Bitsize Functions" for more details on those routines. The example controls a mythical device. The procedure *ReadData* sets up the device's control registers and initiates a transfer from the device into the task's memory. This example is specific to a machine without memory management hardware, such as a small RT-11 system.

```
program Device;                                { example of device control }

const
  Ready = 200B;                                { ready flag }
  ReadBuffer = 1;                              { read data command }

type
  Buffer = packed array [1..100] of char;
  BufferPointer = ^Buffer;

var
  StatusRegister  origin 177316B: integer;
  ControlRegister origin 177314B: integer;
  BufferAddress    origin 177312B: BufferPointer;
  ByteCount       origin 177310B: integer;
  Data: Buffer;    { holds data from device }

procedure ReadData;
begin { ReadData }
  BufferAddress := ref(Data);    { Address for DMA xfer }
  ByteCount := size(Buffer);    { size of buffer }
  ControlRegister := ReadBuffer; { start transfer }
  { Wait for device to complete transfer }
  while (StatusRegister and Ready) = 0 do {wait};
end; { ReadData }

begin { Device }
  ReadData;
end. { Device }
```

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

CHICAGO, ILLINOIS

'Ref' Function

The **ref** function, with a variable argument of type **T**, produces a pointer to that variable with result type **^T** (pointer to **T**). The **dispose** routine cannot always detect attempts to dispose of a pointer generated with this function, and you should not try to do so.

See the example under "Origin Declaration" and under "Loophole Function" for uses of **ref**.

'Size' and 'Bitsize' Functions

Two functions, **size** and **bitsize**, give the programmer information on the space allocated for values of different types. The functions have a single argument, a type identifier.

The function **size** returns the number of bytes that would be allocated for an object of that type by normal variable allocation. The function **bitsize** returns the number of bits that would be allocated for an object of that type as a component of a packed record. This is the actual number of bits required to hold the value.

For example, suppose you had declared a type **Subrange = 0..15** and called the functions **size** and **bitsize**, as in the following example program. The results tell you that two bytes and four bits are allocated for the argument in question.

```
program SizeBitsize;
type
  Subrange = 0..15;

begin
  writeln(size(Subrange));
  writeln(bitsize(Subrange));
end.
```

The program yields these results:

2	_____	2 bytes are allocated
4	_____	4 bits are allocated

These functions are primarily useful when you are interfacing with the operating system or with hardware functions.

See "Origin Declaration" for another example of **size**.

'Loophole' Function

The **loophole** function, by providing a controlled escape from Pascal type rules, allows you to assign variables of one type to a variable of a different type. One use, shown in the **DumpMemory** example following, is to convert a pointer type to an integer type, perhaps to perform pointer arithmetic. The Pascal-2 Debugger, which examines program data, uses **loophole** to look at the stack and compute the values of pointers.

The invocation of **loophole** requires two parameters:

```
loophole(returned-type, expression-to-convert);
```

where *returned-type* is an identifier specifying the data type to be returned by **loophole**, and *expression-to-convert* is an expression of a "compatible" type that is converted to *returned-type*. In this context two types are considered compatible only if they require the same amount of storage (see "Storage Allocation" in the Programmer's Guide), or if they are both non-real scalar types.

The first part of the report deals with the general situation of the country. It is a very interesting and informative study of the country's development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's development.

The second part of the report deals with the specific details of the country's development. It is a very detailed and thorough study of the country's development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's development.

The third part of the report deals with the future of the country's development. It is a very interesting and informative study of the country's development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's development.

Pascal-2 V2.1/RT-11 Language Specification

The result of the `loophole` function is the bit pattern of the second argument, expressed as a value of the type specified in the first argument.

The following program illustrates the compatibility rules that govern the use of `loophole`. The program coerces a real number to an equivalent two-word array of integers representing the two words used to store the real value, then coerces the two-word array back into a real number. The program then coerces an integer in the range 0..4 to a scalar of type `Car`, then coerces the scalar back to an integer. The `loophole(integer, S)` is equivalent to the statement `I := ord(S)`.

```
program Coerce;

type
  Realequiv = array [0..1] of integer;
  Car = (Buick, VW, Datsun, Chevy, BMW); { scalar type }

var
  Re: Realequiv;
  R: real;
  S: Car; { scalar }
  I: integer;

begin { Coerce }
  write('Enter a Real number: ');
  readln(R);
  Re := loophole(Realequiv, R); { coerces real into 2-wd array of integers }
  writeln('Re = ', Re[0]:-8, Re[1]:-8); { 2-wd array printed in octal }
  R := loophole(Real, Re); { coerces 2-wd array back to real }
  writeln('R = ', R);
  write('Enter an integer in range 0..4: ');
  readln(I);
  S := loophole(Car, I); { coerces integer to scalar }
  write('S = ');
  case S of { writes the scalar value }
    Buick: writeln('Buick');
    VW: writeln('VW');
    Datsun: writeln('Datsun');
    Chevy: writeln('Chevy');
    BMW: writeln('BMW');
  end; { case }
  I := loophole(integer, S); { coerces scalar back to integer }
  writeln('I = ', I);
end. { Coerce }
```

When executed, the program yields these results:

```
Enter a Real number: 21567.9
Re =   43650   77715   _____ octal representation of real number (2 words)
R =   2.156790E+04
Enter an integer in range 0..4: 2
S = Datsun
I =      2
```

The only other method of type coercion is to declare a record with variants, using the fact that the compiler overlays storage for different variants. This method makes the same kind of assumptions

Continued from page 10 of 10

The following information was obtained from the records of the Department of the Interior, Bureau of Land Management, regarding the land in question:

1. The land in question is located in the State of California.

2. The land is owned by the State of California.

3. The land is located in the County of Los Angeles.

4. The land is located in the City of Los Angeles.

5. The land is located in the City of Los Angeles.

6. The land is located in the City of Los Angeles.

7. The land is located in the City of Los Angeles.

8. The land is located in the City of Los Angeles.

9. The land is located in the City of Los Angeles.

10. The land is located in the City of Los Angeles.

11. The land is located in the City of Los Angeles.

12. The land is located in the City of Los Angeles.

as the `loophole` function about the compiler's allocation of memory and machine's architecture. However, the `loophole` function has several advantages over variant records:

- No assumption need be made about field allocation in a variant record.
- The compiler checks that the different types are the same size.
- The bypassing of type checking rules is clearly marked (the compiler will flag `loophole` if the `$standard` switch is set). Also, if the code is used with a compiler other than Pascal-2, that compiler should mark `loophole` as an error, and appropriate changes can be made to the code. With variant records, the code might compile but not work.

The following sample program uses the `loophole` function to perform arithmetic on pointers so that a block of the task's memory can be printed.

```

program MDump;

type
  Word = 0..65535;

procedure DumpMemory(Start, Finish: Word);
  type
    Pointer = ^integer;
  var
    P: Pointer;

  begin
    { Dump Memory }
    P := loophole(Pointer, Start);
    while loophole(Word, P) <= Finish do begin
      writeln(loophole(integer, P): -6, ': ', P^: -6);
      P := loophole(Pointer, loophole(Word, P) + 2);
    end;
  end;

begin
  { MDump }
  DumpMemory(1210B, 1220B);
end.
  { MDump }

```

The program yields these results:

```

1210:      6
1212: 101032
1214: 10546
1216: 12746
1220: 177772

```

The next example shows a method to print the address of a variable of any type. The program creates a pointer to the variable, coerces the pointer type into the type used in procedure `WriteAddress`,

Pascal-2 V2.1/RT-11 Language Specification

and prints out the address.

```
program PrintAddress;
```

```
type
```

```
  U_Pointer = 0..65535; { unsigned integers }
```

```
var
```

```
  C: char;
```

```
  R: real;
```

```
  Cptr: ^char;
```

```
  Uptr: U_Pointer;
```

```
  P_Integer: ^integer;
```

```
procedure WriteAddress(A: U_Pointer);
```

```
begin
```

```
  writeln(A: -7); {octal value of address}
```

```
end;
```

```
begin { PrintAddress }
```

```
  C := 'a'; R := 3.54;
```

```
  Cptr := ref(C); { create pointer to char }
```

```
  Uptr := loophole(U_Pointer, Cptr); { coerce pointer into an address }
```

```
  WriteAddress(Uptr); { print out address }
```

```
  WriteAddress(loophole(U_Pointer, ref(R))); { print pointer to real }
```

```
  new(P_Integer);
```

```
  P_Integer^ := 3103;
```

```
  WriteAddress(loophole(U_Pointer, P_Integer)); {print pointer to integer }
```

```
end. { PrintAddress }
```

The program yields these results:

```
11108
```

```
11110
```

```
31436
```


Non-Standard Language Elements

Program Parameters

According to the standard, parameters supplied in the program header indicate external files. Further, the **input** and **output** files must appear in the program header if they are used in the program. The **input** and **output** files are always defined at the global level and may not be redeclared at that level.

With Pascal-2, the program header is not required, and any program parameters are entirely ignored (see "Program Heading" under "Syntax Extensions"). External files are referenced instead by an extended form of **reset** and **rewrite** using a second parameter (a string) giving the external filename (see "External File Access" under "I/O Support Extensions").

Directives

The draft standard treats standard directives such as **forward** as neither an identifier nor a reserved word. Pascal-2 treats the directives **forward**, **external**, **nonpascal** as reserved words. An identifier cannot have the same name as one of these directives.

'Mod' of Negative Numbers

The draft standard states that the divisor must be positive and the operator **mod** must have a non-negative result. That is,

$$0 \leq I \bmod J < J$$

The Pascal-2 compiler generates a divide instruction that gives a negative result if **I** is negative. The standard result can be generated by:

```
Result := I mod J;
if Result < 0 then Result := Result + J;
```

'Eof' Not Accurate For Binary Files

A RT-11 file structure is a sequence of 512-byte blocks. A file containing short records may actually end in the middle of a block, but no information is available as to the end of valid data in the last block, so the **eof** standard function should not be relied upon as accurate. Another method, such as a sentinel record or a record count, should be used to indicate the end of usable data.

Eof is correctly indicated for **text** files.

Structured Types as Function Return Values

Under the standard, functions can return simple data types only (e.g., **integer**, **real**, **char**). With Pascal-2, functions may return structured data types such as **record**, **array** and **set** types in addition to simple types. For example, the function **KeySort**, of structured type, is declared as:

```
function KeySort(Key: KeyType): StructType;
```

where *StructType* is the structured data type of the return value of **KeySort**.

The first part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are in agreement with the experimental facts.

In the second part of the paper, the author discusses the application of the theory of the structure of the atom to the study of the properties of the elements of the periodic table. It is shown that the theory of the structure of the atom can be used to explain the periodicity of the properties of the elements, and that it can be used to predict the properties of the elements which have not yet been discovered.

The third part of the paper is devoted to a discussion of the application of the theory of the structure of the atom to the study of the properties of the compounds of the elements. It is shown that the theory of the structure of the atom can be used to explain the properties of the compounds of the elements, and that it can be used to predict the properties of the compounds which have not yet been discovered.

In the fourth part of the paper, the author discusses the application of the theory of the structure of the atom to the study of the properties of the molecules of the elements. It is shown that the theory of the structure of the atom can be used to explain the properties of the molecules of the elements, and that it can be used to predict the properties of the molecules which have not yet been discovered.

Additional Predefined Functions and Procedures

The Pascal-2 system includes predefined functions and procedures, as allowed by the draft standard. Most of these are grouped according to function in other sections in this guide. This section describes miscellaneous predefined functions and procedures not otherwise described.

Because these procedures and functions are known to the compiler, they need not be declared in the program. The only exception is **timestamp**, which is functionally similar to the other procedures and functions but is not yet predefined. **timestamp** will be predefined in future releases but for now must be declared as an external procedure.

Procedure 'Delete'

The predefined **delete** procedure allows the deletion of a single file that is opened in a Pascal program. **Delete** accepts one argument, the file variable of the file to be deleted. Invoke the procedure with a statement similar to the following:

```
delete(F);
```

Internally, this procedure closes and deletes the file specified by the argument. Your program should not close the file (using **close**) before invoking the **delete** procedure. The run-time error message "can't delete file" results if the file cannot be deleted for some reason. See the example following the discussion of the **rename** procedure.

Procedure 'Rename'

The predefined procedure **rename** allows the renaming of an open file, from within a Pascal program. **Rename** accepts two arguments. The first argument passed to **rename** must be the file variable of the original file name. The second argument must be the file name of the new file. Invoke the procedure with a statement similar to the following.

```
rename(F, 'newfil.txt'); _____ renames F to NEWFIL.TXT  
or:  
NewF := 'newfil.txt';  
rename(F, NewF); _____ renames F to NEWFIL.TXT
```

The second argument may be a constant, a variable, or a literal string. The second argument must contain at least one field. If any fields are omitted from the second argument, the omitted field takes the corresponding value from the original file name. For example, to change the extension only, use a statement similar to this:

```
rename(F, '.mac'); _____ file name is the same; .MAC is the new extension
```

The original file must be open (via **reset**) before **rename** may be called on the file. The renamed file is automatically closed upon completion of the operation.

The following program illustrates the use of the **delete** and **rename** predefined procedures. The program reads a file of weather observations and weeds out duplicate reports, or "dupes." The

[Faint, illegible handwritten text, likely bleed-through from the reverse side of the page.]

Additional Predefined Functions and Procedures

"good" reports are written to a file, which is later renamed. The file of duplicate reports is then deleted.

```
program Dupes;

const
  Climat_File = 'climat.dat';

var
  Data_File: text;      { file of weather observations }
  Dupe_File: text;      { file of duplicate reports }
  Good_File: text;      { file of good reports minus duplicates }

procedure Discard_Dupes(var F, G, H: text);
external;
  { This procedure sorts F, a file of weather observations,
    saving good reports on file G and discarding duplicate
    reports on file H. }

begin
  { Dupes }
  reset(Data_File, 'weax.dat');
  rewrite(Dupe_File, 'dupe.tmp');
  rewrite(Good_File, 'good.dat');
  Discard_Dupes(Data_File, Good_File, Dupe_File); { Weed out the dupes }
  rename(Good_File, Climat_File);  — renames GOOD.DAT to CLIMAT.DAT
  delete(Dupe_File);               — deletes the file of duplicates
end.    { Dupes }
```

Predefined Function 'Time'

The predefined function `time` takes no parameters and returns a real value corresponding to the current time of day. The value `time` is represented in hours after midnight, so that 9:30 a.m. is 9.50 and 1:45 p.m. is 13.75. The resolution of `time` depends on the operating system, but all operating systems provide a resolution of at least one second.

The value returned could be used in header information. (If you wanted the date as well as the time, you would use `timestamp`, described below.) Or you could call `time` at the beginning and end of a text-processing program and write a procedure that calculates the number of lines processed per minute, based on the difference in value returned. Or, because it generates a real number, `time` may be used to "seed" a pseudo-random number generator. The example below returns uses `time` to return the time of day. `Chr(7)` is the "bell" character.

```
program WriteTime;

var
  Hrs, Mins: integer;
  AmPm: packed array[1..2] of char;
```

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

Pascal-2 V2.1/RT-11 Language Specification

```
begin      { WriteTime }
  Mins := Round(time * 60);
  Hrs  := Mins div 60;
  Mins := Mins mod 60;
  if (Hrs < 12) then AmPm := 'AM'
  else if (Hrs = 12) and (Mins = 0)
    then AmPm := 'M ' else AmPm := 'PM';
  write('At the tone the time will be: ');
  write(((Hrs+11) mod 12 + 1):2);
  write(':', Mins div 10:1, Mins mod 10:1, AmPm:3);
  writeln(Chr(7));
end.      { WriteTime }
```

Running the program yields these results:

At the tone the time will be: 11:37 AM <beep>

Procedure 'TimeStamp'

The timestamp procedure provides a way to obtain the date and time from within a Pascal program. Date and time are obtained simultaneously so that they are consistent, even close to midnight.

Timestamp is included in the Pascal-2 library, but the name is not pre-declared by the compiler. You must include a definition similar to:

```
procedure Timestamp(var day, month, year,           { date }
                   hour, min, sec: integer );      { time }

external;
```

The following program prints the date and time using timestamp.

```
program DateTime(output);
var
  Day, Month, Year: Integer;    { date data }
  Hour, Minute, Second: Integer; { time data }

  procedure Timestamp(var Day, Month, Year,           { date }
                     Hour, Min, Sec: Integer );      { time }

    external;

  procedure PrintTwo(N: Integer);
  begin { Print a number on the output file with two digits, including
        leading zeros if needed. The number must be 99 or less }
    write(output, N div 10: 1, N mod 10: 1);
  end; { PrintTwo }
```


Additional Predefined Functions and Procedures

```
begin      { DateTime }
  Timestamp(Day, Month, Year, Hour, Minute, Second);
  PrintTwo(Day);
  case Month of
    1: write(output, '-Jan-');
    2: write(output, '-Feb-');
    3: write(output, '-Mar-');
    4: write(output, '-Apr-');
    5: write(output, '-May-');
    6: write(output, '-Jun-');
    7: write(output, '-Jul-');
    8: write(output, '-Aug-');
    9: write(output, '-Sep-');
    10: write(output, '-Oct-');
    11: write(output, '-Nov-');
    12: write(output, '-Dec-');
  end;
  write(output, Year: 4, ' ');
  PrintTwo(Hour);
  write(output, ':');
  PrintTwo(Minute);
  write(output, ':');
  PrintTwo(Second);
  writeln(output);
end.      { DateTime }
```

The results of the program are:

16-Jun-1983 14:28:31

Journal of the American Medical Association

Published weekly, except during the months of December and January, when it is published bi-weekly.

Subscription prices: Single copies, 10 cents; Six months, \$5.00; One year, \$9.00. In advance. Payment in advance is required. Subscriptions outside the United States add 50 cents per annum for postage. Subscriptions may be ordered through any newsdealer or subscription agent.

Published by the American Medical Association, 535 North Dearborn Street, Chicago, Ill.

Entered as second-class matter, May 2, 1912, under No. 100,000, Post Office at Chicago, Ill., and for mailing at special rate of postage provided for in Act of October 3, 1917, authorized on July 1, 1918.

Error Handling

This section describes the errors defined by the Pascal standard and Pascal-2's handling of them.

Detected Errors

Pascal-2 detects the following errors in all cases:

1. `Ln` or `sqrt` has a negative argument.
2. The integer value returned by `trunc` or `round` lies outside the range `-maxint..maxint`.
3. Integer or real division by zero.
4. The result of a real operation cannot be expressed because of limitations in the floating-point format.
5. No label matches the value of the case index in a `case` statement.
6. The characters being read from a text file do not represent a legal value for the type of variable being read.
7. An attempt to call `get`, `read`, or `readln` when the file has not been `reset` or when `eof` is `true` for that file.
8. An attempt to call `put`, `write`, `writeln`, or `page` when the file has not been rewritten or when `eof` is `false` for that file.
9. A call to `put` when the file variable is undefined.

Pascal-2 detects the following errors under these conditions:

1. The value assigned to a variable or value parameter is not within the declared range of values for that variable. Detected when the `$rangecheck` compiler switch is enabled. (Default.) Not detected when a negative value is assigned to an extended-range variable. See "Extended-Range Arithmetic" for more details.
2. An index expression for an array access is outside the range of the corresponding index type. Detected when the `$indexcheck` switch is enabled. (Default.)
3. A reference through a pointer with a `nil` or undefined value. Reference through a `nil` pointer is detected when the `$pointercheck` switch is enabled. (Default.) Reference through an undefined value is not detected, although many cases will be detected at compile time.
4. In a `for` statement, the initial and final values are not within the range of the controlled variable when the initial value is assigned to the controlled variable. Detected when the `$rangecheck` switch is enabled. (Default.)
5. The calling of `dispose` with a `nil` or undefined parameter. Detected if the parameter is `nil`; detected if the parameter was made undefined by a previous `dispose`. The `dispose` of an undefined pointer is sometimes detected.
6. The result of the `sqr` function is out of range. Detected if the argument type is `real`; undetected if the argument type is `integer`.
7. The result of `chr(x)` is not within the character set. Detected only if a value is assigned to a variable or is passed as a parameter.
8. The result of `succ` or `pred` lies outside the range of the type. Detected only if the value then is assigned to a variable or is passed as a parameter.
9. A `mod` with the right-hand side less than or equal to zero. Detected if the value is zero; otherwise not.

10. Reference to an undefined variable. Undetected in general. However, many simple cases are detected at compile time.
11. A return from a function without a value being assigned to the function. Undetected in general. However, many simple cases are detected at compile time.
12. An attempt to call `put` on a file that was opened with `reset`. Detected except for a file with the `/seek` file control switch specified when the file was opened.

Undetected Errors

Pascal-2 does not detect the following errors:

1. A set value assigned to a set variable or value parameter contains members not in the range of the base type of the set variable.
2. An access to a field in a variant record that is not selected by the current value of the tag-field.
3. A `dispose` of a variable allocated on the heap while there is an active reference to that variable as a variable parameter or in a `with` statement.
4. A change in the value of a file variable by a `get` or `put` while there is an active reference to that variable as a variable parameter or in a `with` statement.
5. Accessing of a variable allocated with `new(p, c1, ..., cn)` as an entire variable, in an assignment or as a parameter.
6. Calling of `dispose(p)` when the value of `p` was created with `new(p, c1, ..., cn)`, or calling of `dispose(p, c1, ..., cn)` with a variable created with `new` and a different set of tag values.
7. The result of an integer operation is incorrect because of overflow.
8. The value of a format expression to a `write` statement is less than 1. Undetected (used in a language extension).

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...
...the ... of ...

Appendix A: Predefined Identifiers

Constants	Functions	Procedures
False	Abs	Break*
Maxint	Arctan	Close*
True	Bitsize*	Delete*
	Chr	Dispose
	Cos	Get
Types	Eof	New
Boolean	Eola	Noicerror*
Char	Exp	Pack
Integer	Icerror*	Page
Real	Icstatus*	Put
Text	Ln	Read
Variables	Loophole*	Readln
Input	Odd	Rename*
Output	Ord	Reset
	Pred	Rewrite
	Ref*	Seek*
	Round	Unpack
	Sin	Write
	Size*	Writeln
	Sqr	
	Sqrt	
	Succ	
	Tinc*	
	Trunc	

Appendix B: Reserved Words

	Function	Packed
And	Goto	Procedure
Array	If	Program
Begin	In	Record
Case	Label	Repeat
Const	Mod	Set
Div	Nil	Then
Do	Nonpascal*	To
Downto	Not	Type
Else	Of	Until
End	Or	Var
External*	Origin*	With
File	Otherwise*	While
For		
Forward		

* Items marked with the asterisk are extensions of standard Pascal.

Continued

Continued

Continued

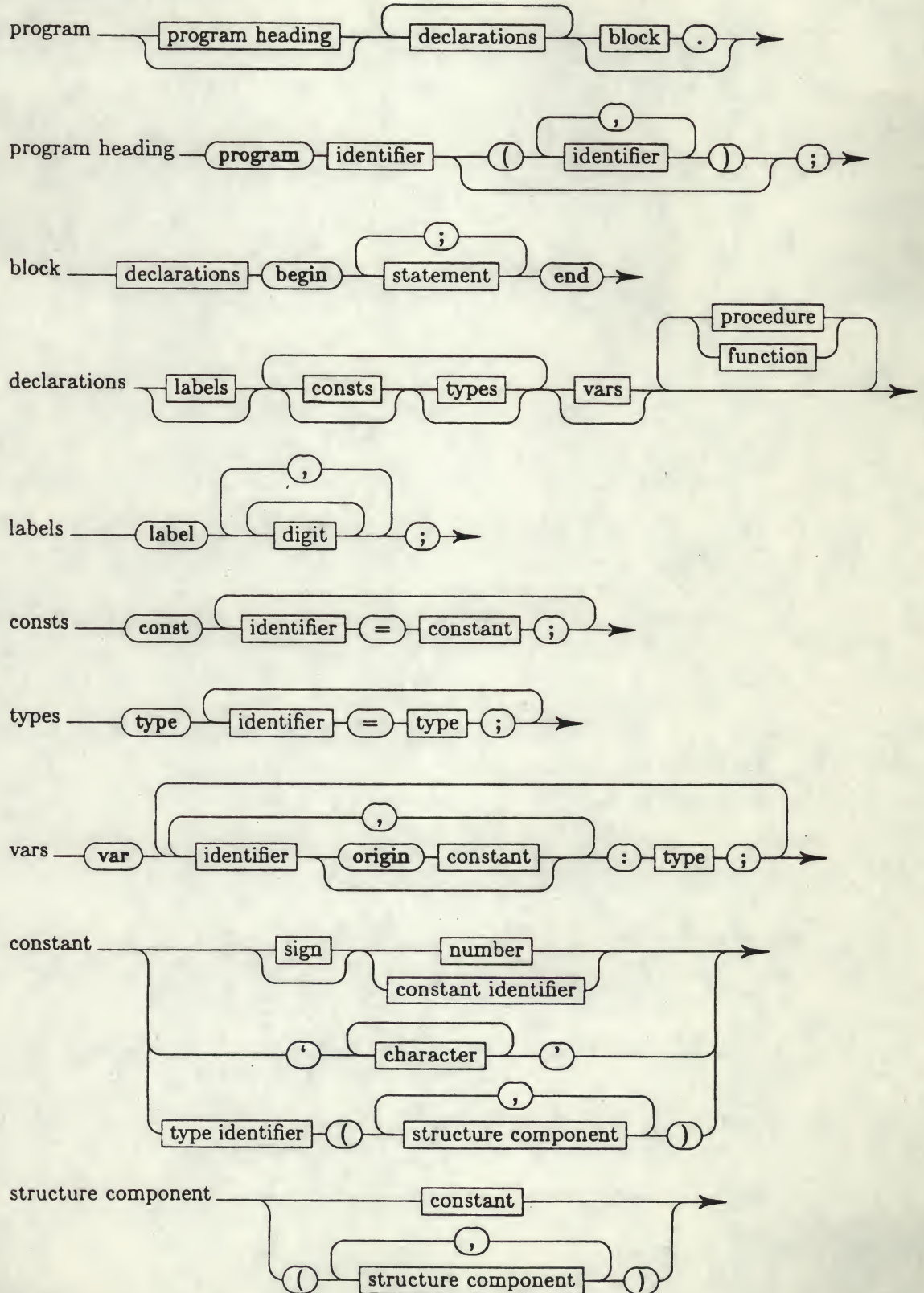
Continued

Continued

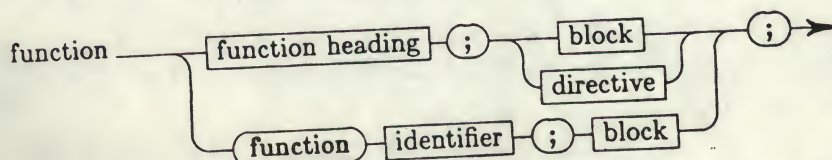
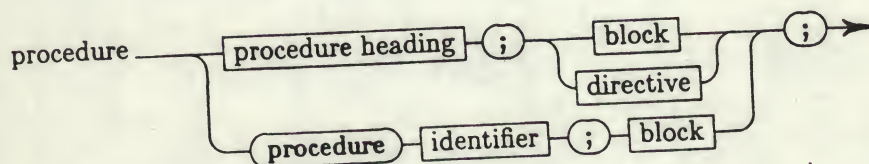
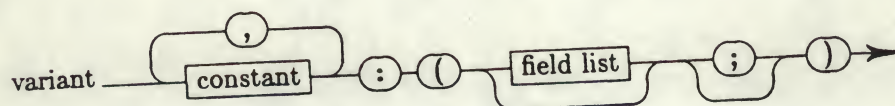
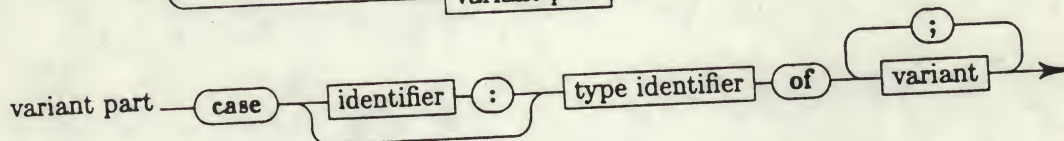
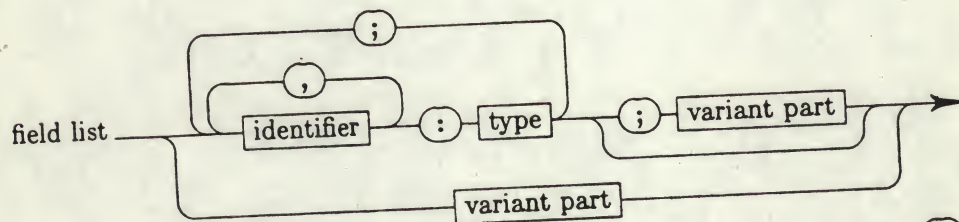
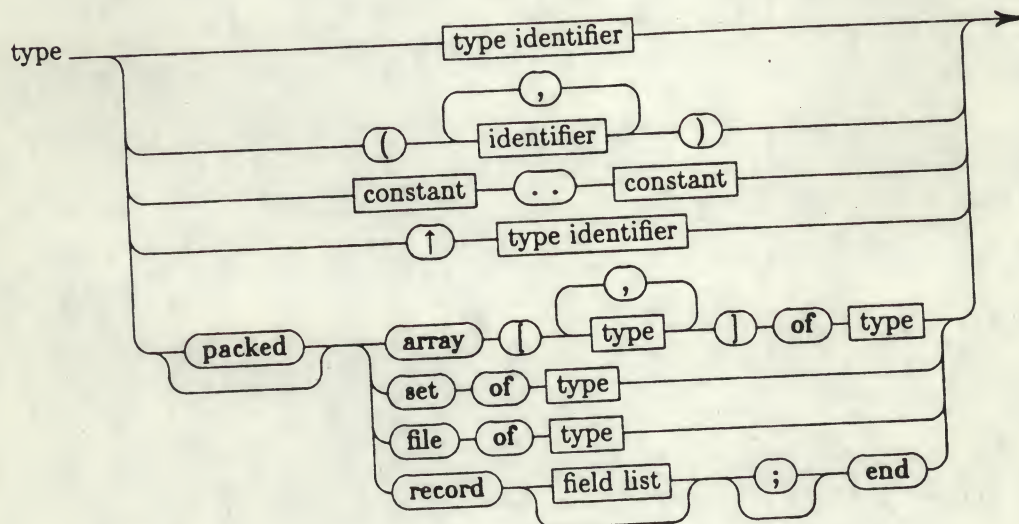
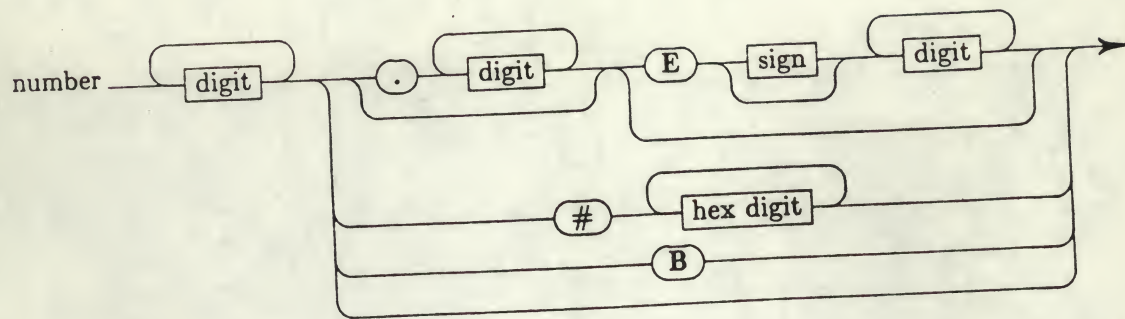
Continued

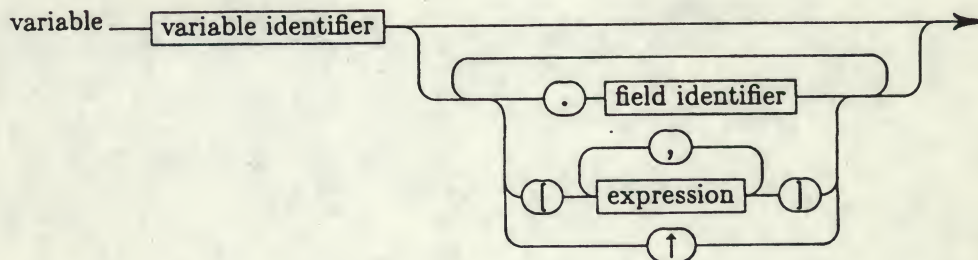
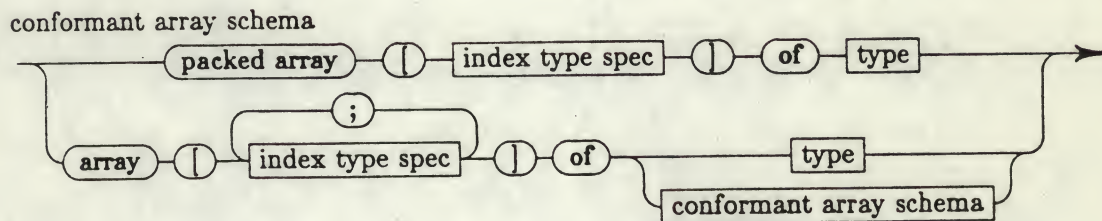
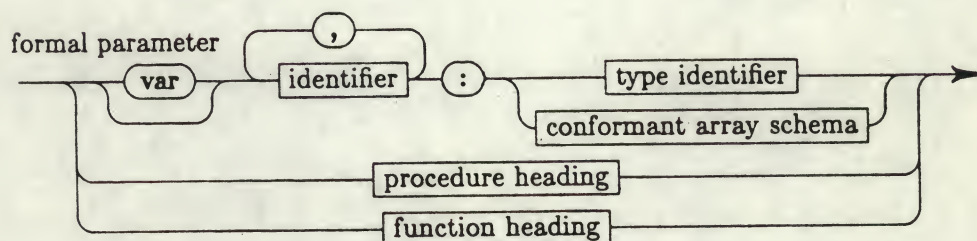
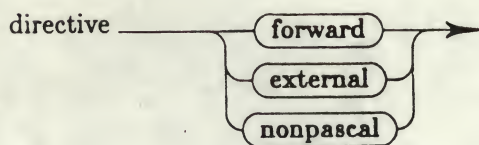
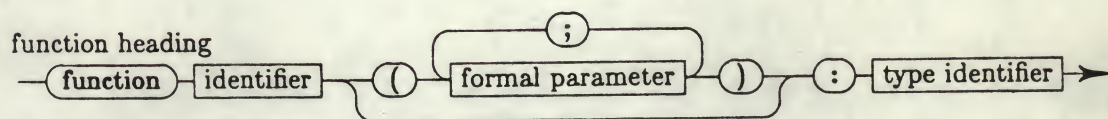
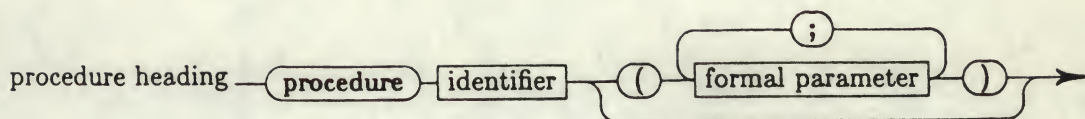
Appendix C: Pascal-2 Syntax

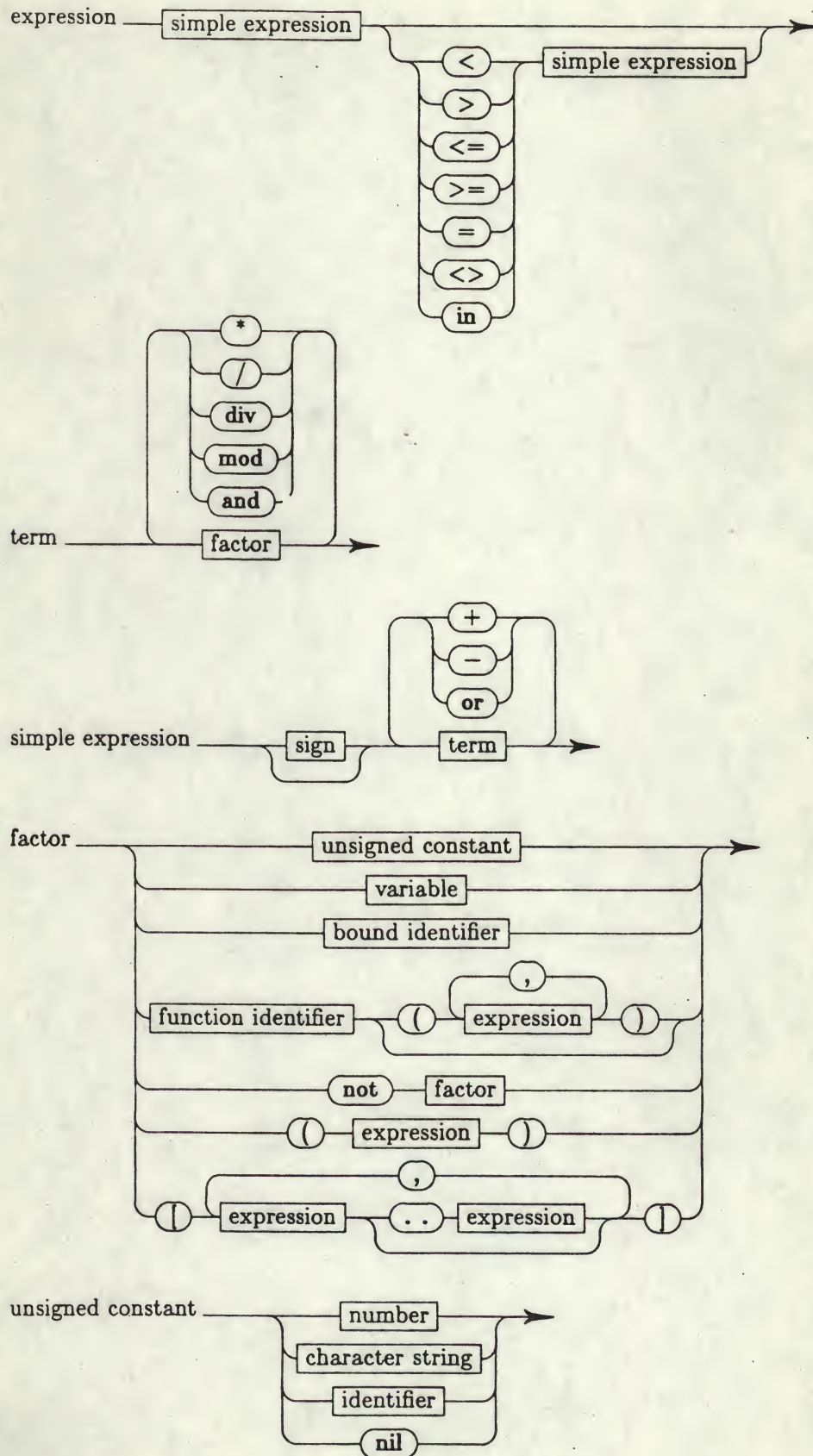
Pascal-2 Syntax Diagrams

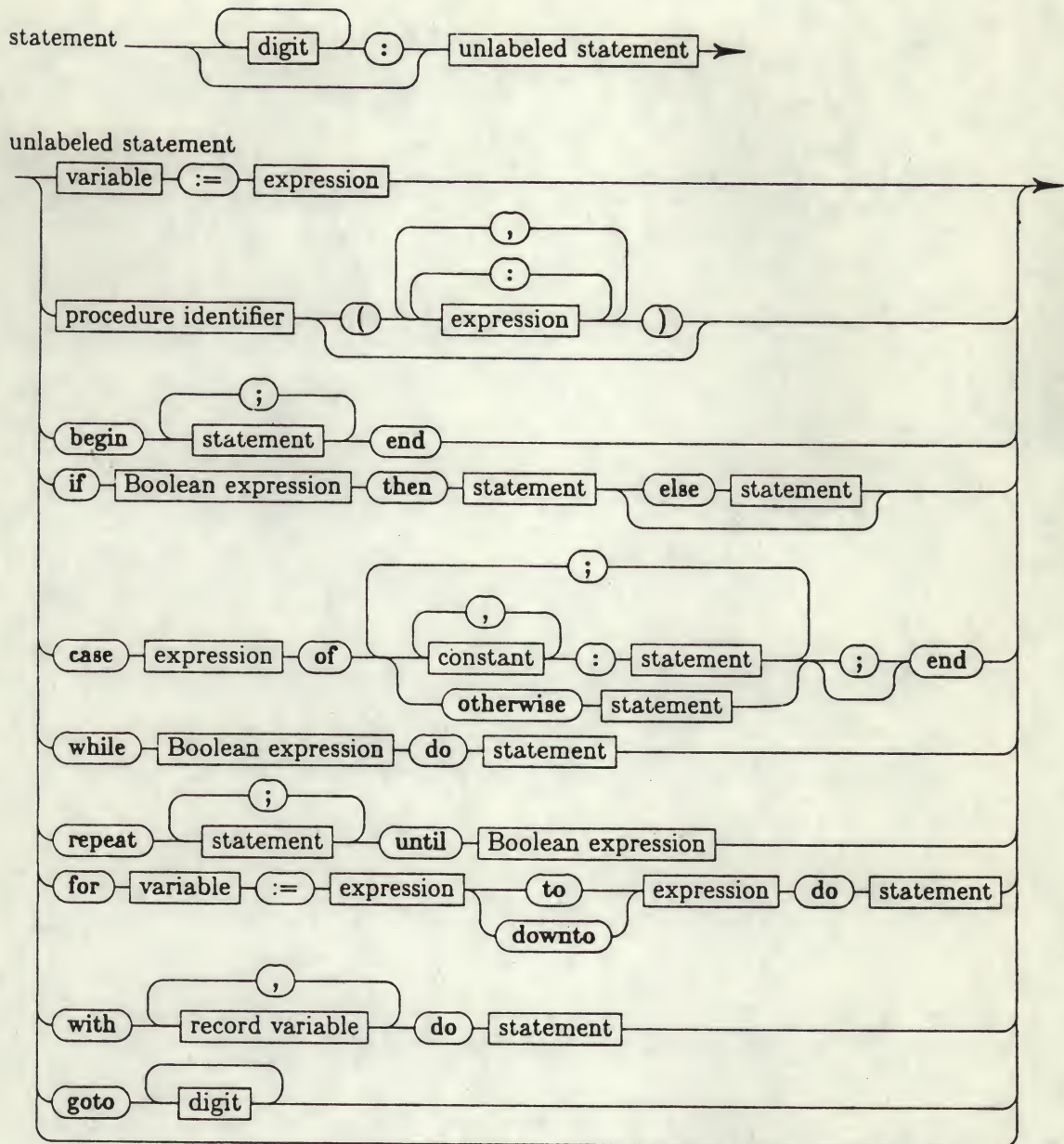


Pascal-2 V2.1/RT-11 Language Specification









Extended Backus-Naur Form

The notation used for describing syntax in this guide is a variant of the Backus-Naur Form (BNF) originally developed to describe the syntax of Algol 60. This particular variant was proposed by Niklaus Wirth ("What Can We Do About the Unnecessary Divergence of Notations for Syntactic Definitions?", *Communications of the ACM*, November 1977, vol. 20, number 11).

A "terminal symbol" is a symbol that actually appears in the language itself. Examples of terminal symbols in Pascal are:

`begin + >=`

Terminal symbols are written in quotes, e.g.: "terminal".

Some terminal symbols are not easily expressed in this way, and these may be represented by comments contained in angle brackets `<>`. For example:

`<any printable character>`

A "nonterminal symbol" is used in the description of the language but does not actually appear in the text of the language. That is, it is used to talk about the language. A nonterminal symbol will stand for some sequence of terminal or nonterminal symbols. Nonterminal symbols are written without quotes. For example:

`identifier, interface-part`

A "production" is a rule specifying which terminal and nonterminal symbols make up another nonterminal symbol. A production is written:

`left-hand-side = right-hand-side .`

The *left-hand-side* is a nonterminal symbol; the *right-hand-side* is some combination of terminal and nonterminal symbols. A production indicates that the *left-hand-side* is made up of the symbols on the *right-hand-side*. A production is terminated with a period.

Within a right-hand-side, the following operators may occur:

(blank) indicates that the two symbols are concatenated. For example:

`lhs = "a" "b" "c" .`

indicates that *lhs* consists of the string `abc`.

| (vertical bar) indicates that the two symbols are alternatives. Concatenation is performed before alternation. For example:

`lhs = "ab" | "cd" .`

indicates that *lhs* consists of one of the strings `ab`, `cd`.

[] (brackets) indicate that the enclosed symbols are optional. For example:

`lhs = "a" ["bc"] "d" .`

indicates that *lhs* consists of one of the strings `abcd`, `ad`.

{ } (braces) indicate that the enclosed symbols are repeated zero or more times. For example:

`lhs = "a" {"b"} "c" .`

Handwritten title or section header.

First paragraph of handwritten text.

Second paragraph of handwritten text.

Third paragraph of handwritten text.

Fourth paragraph of handwritten text.

Fifth paragraph of handwritten text.

Sixth paragraph of handwritten text.

Seventh paragraph of handwritten text.

Eighth paragraph of handwritten text.

indicates that *lhs* consists of any of *ac*, *abc*, *abbc*, *abbbc*,

() (parentheses) are used for grouping as they are in mathematics.

We can now use this notation to describe itself as an example. The productions for *letter*, *digit* and *character* are not given here but are obvious.

syntax = { *production* }.

production = *nonterminal-symbol* "=" *expression* ".".

expression = *term* { "|" *term* }.

term = *factor* { *factor* }.

factor = *nonterminal-symbol* | *terminal-symbol* | "(" *expression* ")"
| "[" *expression* "]" | "{" *expression* "}" .

terminal-symbol = "*" *character* { *character* } "*" | <any comment in angle brackets> .

nonterminal-symbol = *letter* { *letter* | *digit* | "-" }.

Pascal-3 Lexical Description

This set of productions defines the lexical representation of Pascal-2.

Productions that differ from the standard are marked with an asterisk (*).

The case of any alphabetic character is insignificant except in a *character-string*. Lower-case is used in this description.

- 1.* *letter* = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "\$" .
2. *digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
- 3.* *octal-digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .
- 4.* *hexadecimal-digit* = *digit* | "a" | "b" | "c" | "d" | "e" | "f" .
5. *special-symbol* = "+" | "-" | "e" | "/" | "=" | "<" | ">" | "[" | "]" | "(" | ")"
| "." | "," | ":" | ";" | "--" | "@" | "(" | ")" | "<>" | "<=" | ">=" |
| "!=" | "..." | *word-symbol* .
- 6.* *word-symbol* = "and" | "array" | "begin" | "case" | "const" | "div" | "do"
| "downto" | "else" | "end" | "file" | "for" | "function" | "goto" | "if"
| "in" | "label" | "mod" | "nil" | "not" | "of" | "or" | "origin" | "otherwise"
| "packed" | "procedure" | "program" | "record" | "repeat" | "set" | "then"
| "to" | "type" | "until" | "var" | "while" | "with" .
- 7.* *identifier* = *letter* { *letter* | *digit* | "_" }.
8. *bound-identifier* = *identifier* .
- 9.* *directive* = "forward" | "external" | "nonpascal" .
10. *digit-sequence* = *digit* { *digit* }.
- 11.* *unsigned-integer* = [*digit-sequence* "\$"] *hexadecimal-digit-sequence* .
12. *unsigned-real* = (*unsigned-integer* "." *digit-sequence* ["E" *scale-factor*])
fi (*unsigned-integer* "E" *scale-factor*) .

Pascal-2 V2.1/RT-11 Language Specification

- 13.* *nondecimal-integer* = *digit-sequence* "8"
(*hexadecimal-digit* { *hexadecimal-digit* } | *octal-number*) .
- 14.* *octal-number* = *octal-digit* { *octal-digit* } "b" .
- 15.* *unsigned-number* = *unsigned-integer* | *unsigned-real* | *octal-number* .
16. *scale-factor* = *signed-integer* .
17. *sign* = "+" | "-" .
18. *signed-integer* = [*sign*] *unsigned-integer* ;
19. *signed-real* = [*sign*] *unsigned-real* ;
- 20.* *signed-number* = *signed-integer* | *signed-real* | [*sign*] *octal-number* .
21. *label* = *unsigned-integer* ;
22. *character-string* = "1" *string-element* { *string-element* } "1" .
23. *string-element* = "1" | <any printable ASCII character> .
24. *comment* = ("(" | "(")
<any sequence of characters and ends of lines not containing ")" or "*">
(")" | "*") .
- 25.* *lexical-directive* = "%include" "1" *file-name-string* "1" ";" | "%page" ";" .

Pascal-2 EBNF Syntax

This set of productions defines the syntax for the language accepted by the Pascal-2 compiler, including all extensions.

This section is to be interpreted in conjunction with the lexical description of the language.

Productions are based on those in the ISO standard. Where the language accepted by the Pascal-2 compiler differs from this standard, the production is marked with an asterisk (*).

- 1.* *program* = [*program-heading*] { *label-declaration-part*
| *constant-definition-part* | *type-definition-part*
| *variable-declaration-part* | *routine-declaration* } [*body* "."] .
2. *program-heading* = "program" *identifier* ["(" *program-parameters* ")"] ";" .
3. *program-parameters* = *identifier* { "," *identifier* } .
4. *block* = *declarations* *body* .
- 5.* *declarations* = [*label-declaration-part*] [*constant-definition-part*
| *type-definition-part*] [*variable-declaration-part*] { *routine-declaration* } .
6. *label-declaration-part* = "label" *label* { "," *label* } ";" .
7. *constant-definition-part* = "const" *constant-definition* { ";" *constant-definition* } ";" .
8. *constant-definition* = *identifier* "=" *constant* .
- 9.* *constant* = ([*sign*] (*unsigned-number* | *identifier*))
| *character-string* | *structured-constant* .
- 10.* *structured-constant* = *structured-type-identifier* *constant-component-list* .
- 11.* *constant-component-list* = "(" *constant-component* { "," *constant-component* } ")" .
- 12.* *constant-component* = *constant* | *constant-component-list* .

The first part of the paper discusses the importance of maintaining accurate records of all transactions. It is essential for the company to have a clear and concise system in place to ensure that all data is properly recorded and stored. This will allow for easy access and retrieval of information when needed.

The second part of the paper focuses on the importance of regular communication and collaboration between all team members. It is crucial for everyone to stay informed about the company's goals and objectives, as well as the progress of various projects. Regular meetings and updates will help to ensure that everyone is working towards the same goals and that any issues are identified and resolved quickly.

The third part of the paper discusses the importance of maintaining a high level of security for all company data. It is essential to implement strong security measures to protect sensitive information from unauthorized access or theft. This includes using secure communication channels, implementing strong passwords, and regularly updating security software.

The fourth part of the paper discusses the importance of maintaining a high level of transparency and accountability. It is essential for the company to be open and honest about its operations and financial performance. This will help to build trust with stakeholders and ensure that everyone is held accountable for their actions.

The fifth part of the paper discusses the importance of maintaining a high level of flexibility and adaptability. The business environment is constantly changing, and it is essential for the company to be able to quickly respond to new challenges and opportunities. This requires a flexible and adaptable mindset, as well as a willingness to try new things and take risks.

The sixth part of the paper discusses the importance of maintaining a high level of innovation and creativity. It is essential for the company to be able to think outside the box and come up with new and innovative solutions to problems. This requires a culture of innovation and creativity, as well as a willingness to experiment and take risks.

The seventh part of the paper discusses the importance of maintaining a high level of customer satisfaction. It is essential for the company to be able to provide high-quality products and services that meet the needs and expectations of its customers. This requires a focus on customer service and a willingness to go above and beyond to ensure that customers are satisfied.

The eighth part of the paper discusses the importance of maintaining a high level of employee satisfaction. It is essential for the company to be able to provide a positive and supportive work environment for its employees. This requires a focus on employee well-being and a willingness to listen to and address employee concerns.

The ninth part of the paper discusses the importance of maintaining a high level of social responsibility. It is essential for the company to be able to contribute positively to society and the environment. This requires a focus on ethical business practices and a willingness to support social and environmental causes.

13. *type-definition-part* = "type" *type-definition* { ";" *type-definition* } ";" .
14. *type-definition* = *identifier* "=" *type* .
15. *type* = *identifier* | *enumerated-type* | *subrange-type* | *set-type*
| *array-type* | *record-type* | *file-type* | ("^" | "o" *identifier*) .
16. *enumerated-type* = "(" *identifier* { "," *identifier* } ")" .
17. *subrange-type* = *constant* ".." *constant* .
18. *set-type* = ["packed"] "set" "of" *type* .
19. *array-type* = ["packed"] "array" "[" *type* { "," *type* } "]" "of" *type* .
20. *record-type* = ["packed"] "record" *field-list* [";"] "end" .
21. *field-list* = (*fixed-part* [";" *variant-part*]) | *variant-part* .
22. *fixed-part* = *record-section* { ";" *record-section* } .
23. *record-section* = *identifier* { "," *identifier* } ":" *type* .
24. *variant-part* = "case" [*identifier* ":"] *identifier* "of" *variant* { ";" *variant* } .
25. *variant* = *constant* { "," *constant* } ":" "(" [*field-list*] [";"] ")" .
26. *file-type* = ["packed"] "file" "of" *type* .
27. *variable-declaration-part* = "var" *variable-declaration* ";" { *variable-declaration* ";" } .
28. *variable-declaration* = *var-specification* { ";" *var-specification* } ":" *type* .
29. *var-specification* = *identifier* ["origin" *constant*] .
30. *routine-declaration* = (*procedure-declaration* | *function-declaration*) ";" .
31. *procedure-declaration* = (*procedure-heading* ";" *block*)
| (*procedure-heading* ";" *directive*) | (*procedure-ident* ";" *block*) .
32. *procedure-heading* = "procedure" *identifier* [*parameter-list*] .
33. *procedure-ident* = "procedure" *identifier* .
34. *function-declaration* = (*function-heading* ";" *block*)
| (*function-heading* ";" *directive*) | (*function-ident* ";" *block*) .
35. *function-heading* = "function" *identifier* [*parameter-list*] ":" *identifier* .
36. *function-ident* = "function" *identifier* .
37. *parameter-list* = "(" *parameter-section* { ";" *parameter-section* } ")" .
38. *parameter-section* = (["var"] *identifier* { "," *identifier* } ":" (*identifier*
| *conformant-array-schema*)) | *procedure-heading* | *function-heading* .
39. *conformant-array-schema* = *packed-conformant-array-schema*
| *unpacked-conformant-array-schema* .
40. *packed-conformant-array-schema* = "packed" "array"
"[" *index-type-specification* "]" "of" *type* .
41. *unpacked-conformant-array-schema* = "array" "[" *index-type-specification*
{ ";" *index-type-specification* } "]" "of" (*type* | *conformant-array-schema*) .
42. *index-type-specification* = *bound-identifier* ".." *bound-identifier* ":" *type* .
43. *body* = *compound-statement* .

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

Pascal-2 V2.1/RT-11 Language Specification

44. *statement* = [*label* ":"]
 [*assignment* | *procedure-call* | *compound-statement* | *if-statement*
 | *case-statement* | *while-statement* | *repeat-statement*
 | *for-statement* | *with-statement* | *goto-statement*] .
45. *assignment* = *variable* " : = " *expression* .
46. *procedure-call* = *identifier* [*arg-list* | *write-arg-list*] .
47. *arg-list* = "(" *expression* { "," *expression* } ")" .
48. *write-arg-list* = "(" *write-arg* { "," *write-arg* } ")" .
49. *write-arg* = *expression* [":" *expression* [":" *expression*]] .
50. *compound-statement* = "begin" *statement* { ";" *statement* } "end" .
51. *if-statement* = "if" *expression* "then" *statement* ["else" *statement*] .
- 52.* *case-statement* = "case" *expression* "of" [*case-element* { ";" *case-element* }] [":"]
 ["otherwise" *statement* [";"]] "end" .
53. *case-element* = *constant* { "," *constant* } ":" *statement* .
54. *while-statement* = "while" *expression* "do" *statement* .
55. *repeat-statement* = "repeat" *statement* { ";" *statement* } "until" *expression* .
56. *for-statement* = "for" *identifier* " : = " *expression* ("to" | "downto") *expression*
 "do" *statement* .
57. *with-statement* = "with" *expression* { "," *expression* } "do" *statement* .
58. *goto-statement* = "goto" *label* .
59. *expression* = *simple-expression* [*relational-operator* *simple-expression*] .
60. *relational-operator* = "<" | ">" | "<=" | ">=" | "=" | "<>" | "in" .
61. *simple-expression* = [*sign*] *term* { *adding-operator* *term* } .
62. *adding-operator* = "+" | "-" | "or" .
63. *term* = *factor* { *multiplying-operator* *factor* } .
64. *multiplying-operator* = "*" | "/" | "div" | "mod" | "and" .
65. *factor* = *unsigned-constant* | *variable* | *function-call* | "not" *factor*
 | "(" *expression* ")" | *bound-identifier* | ("[" | "(")
 [*member-designator* { "," *member-designator* }] ("]" | ")") .
66. *unsigned-constant* = *unsigned-number* | *string* | *identifier* | "nil" .
67. *function-call* = *identifier* [*arg-list*] .
68. *variable* = *identifier* | *variable* ("[" | "(") *expression* { "," *expression* }
 ("]" | ")") | *variable* ("~" | "o") | *variable* "." *identifier* .
69. *member-designator* = *expression* ["." *expression*] .

[The text on this page is extremely faint and illegible. It appears to be a handwritten letter or document, possibly containing names and dates, but the characters are too light to transcribe accurately.]

Pascal-2 V2.1/RT-11 Debugger Guide

Debugging tools help uncover "run-time" errors—errors in a program's execution—that cannot be caught during compilation. For example, a procedure may generate an incorrect number of loops or make a legal but unintended change in the value of a variable. The Pascal-2 Debugger lets you control a program's execution interactively; you may suspend execution at particular statements to examine or modify the values of variables, or you may execute statements one at a time to trace the actions leading to an incorrect result.

When called, the Pascal-2 Debugger keeps track of all constants, variables, local procedures and functions and all standard and user-defined data types. The Debugger can show what's happening to data and allow you to change the data as the program executes. You can display the original source text of your program for immediate identification of context, and you can access and debug external procedures and functions called by the main program. (See "Debugging External Modules" at the end of this guide for details.) The Debugger also traps errors by halting execution of a program at the point of breakdown and identifying the last statement executed. Taken together, these features allow you to trouble-shoot a program until you have detected and corrected any errors.

This guide serves as an introduction to the Pascal-2 debugging process and as a comprehensive resource for operation of the Pascal-2 Debugger. The guide provides:

- An overview of the Pascal-2 debugging environment.
- Detailed descriptions of the Debugger commands.
- A tutorial that demonstrates the context in which Pascal-2 Debugger commands are most frequently used.
- An explanation of how external modules are debugged.
- A one-page summary of Debugger commands.

A word of warning before beginning: specifying the debugging option causes the compiler to include a call to the Debugger before each procedure and statement, which substantially increases the size of your program. The object module created by the compiler contains extra code to locate statements and procedures in your program. Moreover, introducing the Debugger turns off optimizations that would interfere with debugging. The compiler normally folds similar statements into one section of code and optimizes the usage of some variables by keeping their values in registers on the stack temporarily. These optimizations would keep the Debugger from setting breakpoints in statements and from changing the values of variables while your program was running—both of which are important debugging facilities. A little bit of memory is saved during use of the Debugger by disabling the procedure walkback—this happens automatically when the Debugger is implemented—but in general, the overhead involved in using the Pascal-2 Debugger is about one word per Pascal statement and about six words per procedure. Code returns to its normal size once you correct any problems and recompile your program without a call to the Debugger. (See "Overlays" in this guide regarding what to do if the program grows too large.)

THE HISTORY OF THE UNITED STATES

The first part of the history of the United States is the period from the discovery of the continent by Christopher Columbus in 1492 to the establishment of the first permanent settlements in 1607. This period is characterized by the exploration of the continent by Spanish, French, and English explorers, and the establishment of the first permanent settlements in the eastern part of the continent.

The second part of the history of the United States is the period from 1607 to 1776. This period is characterized by the growth of the colonies, the struggle for independence from Britain, and the establishment of the United States as a new nation. The colonies were established by English, French, and Dutch settlers, and they grew into a powerful nation that was able to win its independence from Britain in 1776.

The third part of the history of the United States is the period from 1776 to 1865. This period is characterized by the American Revolution, the War of 1812, and the Civil War. The American Revolution was a struggle for independence from Britain, and the War of 1812 was a conflict between the United States and Britain. The Civil War was a conflict between the Union and the Confederacy, and it ended in 1865 with the Union's victory.

The fourth part of the history of the United States is the period from 1865 to 1945. This period is characterized by the Reconstruction era, the Gilded Age, and the Progressive Era. The Reconstruction era was a period of rebuilding the South after the Civil War, and the Gilded Age was a period of rapid economic growth and industrialization. The Progressive Era was a period of reform and social change.

The fifth part of the history of the United States is the period from 1945 to the present. This period is characterized by the Cold War, the Vietnam War, and the Civil Rights Movement. The Cold War was a period of tension between the United States and the Soviet Union, and the Vietnam War was a conflict between the United States and North Vietnam. The Civil Rights Movement was a struggle for equality for African Americans, and it led to the passage of the Civil Rights Act of 1964.

Including the Pascal-2 Debugger in Your Program

The debug compilation switch invokes the Pascal-2 Debugger. (See the User Guide for details on compilation switches.) Using the debug switch in your compilation command automatically generates a formatted listing file, with an .LST extension, in the same directory as the output file. The Debugger reads this listing file to display the source lines when statements are identified. The Debugger can use only the listing file produced by the debug switch.

The debug switch also causes the compiler to create symbol table and statement map files in the same directory as the output file. The symbol table file (extension .SYM) describes the constants, types, variables, and the memory layout of variables. The symbol table file also contains information about each procedure and function local to the compilation unit. The statement map file (extension .SMP) contains a map of the location of the statements and their position in the listing. Both files are in binary form and are not readily examined by users.

.R PASCAL
•ROTAT/DEBUG

.LINK ROTAT.SY:PASCAL

Identifying Pascal Statements

Remember, the debug switch automatically generates a listing file. As the example ROTAT.LST shows, a listing file has two columns of numbers. The leftmost column lists the line numbers in the source file. The second column contains the number of each statement in the program, beginning with '1' for each procedure or function. These numbers identify points where you may set breakpoints to interrupt program execution. In ROTAT.LST, several lines accessible to the Debugger have been labeled by procedure name and statement number. As shown, statements in the main body of the program are considered to be in the procedure MAIN. All Pascal programs begin executing at MAIN, 1.

You should have a printout of the listing file as reference when you begin a debugging session, or you may use the L command to list parts of the program while you are debugging.

Example:

Pascal-2 RT11 SJ V2.1D 9-Feb-84 7:06 AM Site #1-1 Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
ROTAT/DEBUG

```

Line Stmt
1      program Rotat; { rotate an array of numbers }
2
3      const Arraylen = 7;
4
5      type  Index = 1..Arraylen;  Element = 0..10;
6           Numbers = array [index] of Element;
7
8      var   I: Index;  N: Numbers;  Left, Right: Index;
9
10     procedure Rotate(First, Last: Index;
11                      var A: Numbers);
12         var I: Index;
13
14     1     begin
15     2         for I := First to Last do
16     3             A[I] := A[I + 1];           ROTATE, 3
17     4             A[Last] := A[First];       ROTATE, 4
18     5             write('Rotated ', first: 1, ' thru ', last: 1, '=');
19         end;
20
21     1     begin { main program }           MAIN, 1
22     2         for I := 1 to Arraylen do
23     3             begin N[I] := I; write(I: 2); end;
24     5             writeln; write('Left,Right? ');
25     7             readln(Left, Right);
26     8             I := 4;
27     9             Rotate(Left, Right, N);
28    10         for I := 1 to Arraylen do
29    11             write(N[I]: 2);
30    12         writeln
31     end.

```

*** No lines with errors detected ***

ROTAT.PAS is worth studying for a moment because it appears frequently throughout the remainder of this guide. The program prints an array of seven integers, then prompts you, asking for a starting and ending point in the array. Once the two input numbers have been entered, the program is supposed to rotate that section of integers to the left, with the left digit replacing the right. In its current form, the program compiles without problem, but as is shown in several of the following sections, its execution encounters numerous run-time errors.

1962

THE UNIVERSITY OF CHICAGO
DEPARTMENT OF CHEMISTRY

RECEIVED

FROM

DATE

BY

REMARKS

INITIALS

SIGNATURE

DATE

BY

REMARKS

INITIALS

SIGNATURE

DATE

BY

REMARKS

INITIALS

SIGNATURE

DATE

BY

REMARKS

INITIALS

SIGNATURE

DATE

BY

REMARKS

INITIALS

SIGNATURE

Pascal-2 V2.1/RT-11 Debugger Guide

Controlling the Debugger

Debugger takes control of the program, enters the command mode, and prompts with a right brace '}' symbol. (This may print on upper-case-only terminals as the right bracket ']' character.)

```
.RUN ROTAT
```

```
Pascal Debugger V3.00 -- 29-Nov-1983
```

```
Debugging program ROTAT
```

```
}
```

You control the Debugger through single-character commands that generally take one of two forms, depending on whether or not the command accepts parameters:

- } *single-character command*
- } *single-character command (parameter(s))*

Debugger commands and their parameters may be typed in either upper or lower case.

Command Syntax

In general, Debugger commands are used for controlling breakpoints, program execution, program tracking, data, and for displaying information about the data being maintained by the Debugger. Debugger commands can be stored in series and executed at designated locations within a program. Such locations are known as breakpoints and are specified by the break command. At any breakpoint, you may enter as many stored commands as fit on a single line. Any Debugger command may appear in a stored command and certain utility commands, described later, allow macros to be defined that let you store combined commands.

As an example of the general use of Debugger commands and the syntax for writing stored commands, look at the following command line. The line begins with the single-character command B followed by the parameter ROTATE,3. These direct the Debugger to set a breakpoint at statement 3 in procedure ROTATE. Next, a stored command is used to instruct the Debugger to write (W) the values of the variables I and A[I]. Stored commands are specified by placing them within angle brackets (<...>) after a break command and separating them by semicolons.

```
} B(ROTATE,3) <W(I); W(A[I])>
```

The Debugger accepts any of the single-character commands defined in the following sections. Numeric parameters in these sections are indicated by 'n', as in the command S(n). A summary of the Debugger commands is given in Appendix A at the end of this guide, and the ? (question mark) command prints a similar list on your terminal screen.

Handwritten text at the top of the page, possibly a header or introductory paragraph.

Main body of handwritten text, consisting of several paragraphs.

Handwritten text at the bottom of the page, possibly a signature or footer.

Exiting and Stopping the Debugger

To exit from the Debugger at the prompt, give the command **Q** (quit), or type a Control-Z (**^Z**), or type Control-C (**^C**) twice in a row. A single Control-C (**^C**) typed during program execution stops the Debugger, thus permitting you to break into "infinite loops" in your program.

Selective Debugging

For certain large programs, you may wish to selectively debug portions of a program in order to speed up the debugging process or to reduce the amount of memory overhead created by the Debugger. You can edit your program to turn off generation of debugging information around procedures that have already been tested and debugged by using the embedded directives **\$nocodebug** and **\$debug**. To turn off debugging, place the directive **\$nocodebug** before the procedure definition and the directive **\$debug** after the procedure. **\$nocodebug** and **\$debug** are effective only when the **/debug** switch is first specified in the compilation command line. Otherwise they are ignored by the compiler. (See the User Guide for further details on embedded directives.)

1890-1891

The first of the year was a very dry one, and the crops were much injured by the drought. The weather was very hot, and the crops were much injured by the drought.

The second of the year was a very wet one, and the crops were much injured by the drought. The weather was very hot, and the crops were much injured by the drought. The third of the year was a very dry one, and the crops were much injured by the drought. The weather was very hot, and the crops were much injured by the drought.

Breakpoint Commands

Breakpoint commands allow you to set or remove breakpoints when your program reaches a certain point in execution or when a specified variable in your program changes value. Breakpoints allow you to interrupt the program in order to execute other Debugger commands.

B, B(): Control Breakpoints

A program control breakpoint is identified by two items: a block name (procedure, function, or **MAIN**), and a statement number within that block. For example, **ROTATE,3** identifies the third statement in the procedure **ROTATE**. Statements are sequentially numbered within each block. Statement numbers are listed in the second column of the program listing produced by the **debug** switch.

The **B(block,stmtnum)** command sets a control breakpoint within the block named *block* at the statement numbered *stmtnum*. When the breakpoint is reached, your program is interrupted before execution of the named statement, the breakpoint is identified, and the Pascal source line is displayed. The Debugger then accepts commands.

These may be interactive commands (from your terminal) or stored commands executed automatically. Any Debugger command may be stored for execution at a breakpoint. Stored commands are executed before interactive commands. If the stored commands direct the Debugger to resume execution, the program continues without waiting for an interactive command.

You may interrupt the program at any time with a Control-C (^C). This command stops the program and identifies the point of interruption as if you had set a control breakpoint.

NOTE

If you type a Control-C (^C) while the program is awaiting input for a **real** or an **integer** at a **read** or **readln** statement, the Control-C (^C) does not take effect until after you have completed the input request.

A run-time error or program termination also causes a control breakpoint after the error message or termination status is displayed. You may set any number of control breakpoints. (The program executes more slowly if you define many.)

To set breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

You may remove a breakpoint in two ways. The **B** command with no parameters deletes the breakpoint that most recently stopped the program. Otherwise, the **K** command described next may be used. (For uses of the **B** command, see the example listed after the **C** command.)

K, K(): Killing of Breakpoints

The **K(block,stmtnum)** command deletes the breakpoint specified by its parameter; the **K** command with no parameters removes all breakpoints. (See the example after the **C** command.) To remove breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

THE UNIVERSITY OF CHICAGO
CHICAGO, ILLINOIS

V, V(): Data Breakpoints (Variables)

The data breakpoint facility (also called the "watched variable" command) causes an immediate breakpoint when the value of a specified variable is changed. The **V(variable)** command sets a data breakpoint, with *variable* indicating the variable to be monitored. When the value of the variable is changed, the Debugger prints both the old and new values and interrupts program execution for commands. Like control breakpoints, data breakpoints may have stored commands that are automatically executed when the breakpoint is triggered. A list of the stored commands, separated by semicolons, is enclosed in angle brackets after the watched variable command: **V(variable)< ... >**.

The **V** command monitors a variable of any type, but only the first 32 bytes of data is watched. You may watch any number of variables. (The program executes slowly if you set many.) For variables defined locally to a procedure, the watch command can either be set from within the procedure or through use of the **E** command defined later in this guide.

```

} B(ROTATE,1)<V(A[6])>      variable watch set within ROTATE
} G                          begin execution
Left,Right? 2 6             input to ROTAT
Breakpoint at ROTATE,1      begin
} G                          continue execution
The value of "A[6]" was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4      A[Last] := A[First];
}

```

If a local variable is being monitored and the associated block is completed, the Debugger removes the breakpoint and displays a message that the variable no longer exists.

```

Breakpoint at ROTATE,1      begin      previously set breakpoint
} L                          lists statements of procedure ROTATE
14  1      begin
15  2      for I := First to Last do
16  3      A[I] := A[I + 1];
17  4      A[Last] := A[First];
18  5      write('Rotated ', first: 1, ' thru ', last: 1, '=');
19      end;
} V(A[2])                  variable watched within ROTATE
} G
The value of "A[2]" was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 2
New value: 3
Breakpoint at ROTATE,3      A[I] := A[I + 1];
} G
Watch terminated for "A[2]" Value did not change.
Rotated 2 thru 6= 1 3 4 5 6 3 7      indicates a run-time error
}

```

This example gives us our first indication of a problem in the program ROTAT.PAS. The correct rotation for the starting and ending points (2,6) should read 1 3 4 5 6 2 7 not 1 3 4 5 6 3 7. Correction of the problem is explained in the section "Stepping Through a Debugging Session" later in this guide.

The **V** command without parameters removes all data breakpoints. It is not possible to remove individual data breakpoints.

1890

1890

My dear Mr. [Name],
I have just received your letter of the 10th inst. and am
glad to hear that you are well. I am also well and hope
this letter finds you the same. I have been thinking of
writing to you for some time but have been so busy that I
could not find time. I am now at home and have some
time to spare. I hope to hear from you again soon.

I have been thinking of writing to you for some time
but have been so busy that I could not find time. I am
now at home and have some time to spare. I hope to
hear from you again soon.

I have been thinking of writing to you for some time
but have been so busy that I could not find time. I am
now at home and have some time to spare. I hope to
hear from you again soon.

I have been thinking of writing to you for some time
but have been so busy that I could not find time. I am
now at home and have some time to spare. I hope to
hear from you again soon.

I have been thinking of writing to you for some time
but have been so busy that I could not find time. I am
now at home and have some time to spare. I hope to
hear from you again soon.

Execution Control Commands

Execution control commands provide the means to monitor and control the flow of the program. The commands initiate, interrupt, or continue execution.

G: Go

The G (Go) command begins executing the program at **MAIN,1** and may be used at any point in the program to restart it.

See the example after the C command.

C, C(): Continue Execution

The C (Continue) command resumes program execution from the current breakpoint.

If you set a breakpoint inside a loop, you may use the C(n) command to let the statement at the breakpoint execute n times. For instance, you may set a breakpoint at **COUNT,10** inside a loop structure. When the Debugger stops at that breakpoint, you may give the command C(6) to let the loop iterate six times before the program stops again at **COUNT,10**. Each breakpoint has its own counter, which is independent of the counters for other breakpoints.

The C command functions like the G command to begin executing the program if you are at the start of the program.

If you use the C command after the program has terminated, you receive an error message telling you to use the G command to restart the program.

Examples of the B, K, D, G and C Commands

.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

```

} L(main,8,2) _____ List 8th statement of MAIN, 2 lines
  26      8      I := 4;
  27      9      Rotate(Left, Right, N);
} B(main,9)<W('I=' ,1);C>
} B(Rotate,3)<W('In rotate, I=' ,1)>
} G
  1 2 3 4 5 6 7
Left,Right? 2 6 _____ input to ROTAT
Breakpoint at MAIN,9 Rotate(Left, Right, N);
I= 4
Breakpoint at ROTATE,3 A[I] := A[I + 1];
In rotate, I= 2
} D _____ display breakpoints

```

Breakpoints

```

ROTATE,3 A[I] := A[I + 1];
      <W('In rotate, I=' ,I)>

```

```

MAIN,9 Rotate(Left, Right, N);
      <W('I=' ,I);C>

```

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10

10/10/10


```

} W(I): C(2): W(I)
2
Breakpoint at ROTATE,3  A[I] := A[I + 1];
In rotate, I= 4
4
} K(Rotate,3) _____ kill specified breakpoint
} G _____ continue execution
Rotated 2 thru 6= 1 3 4 5 6 3 7 _____ indicates run-time error
} D _____ display breakpoints

```

Breakpoints

```

MAIN,0 Rotate(Left, Right, N);
      <W('I',I);C>

```

```

} K _____ kill all breakpoints
} D _____ (no breakpoints to display)
} Q _____ quit the Debugger

```

S, S(): Step to Next Statement

The S (Step) command executes the next statement of the program. The S(*n*) command executes *n* statements without interruption. If a statement being "stepped" calls another procedure or function, that new procedure or function also is executed one step at a time.

See the example after the P command.

P, P(): Proceed to Next Statement

The P (Proceed) command executes the next statement at the current level of the program. P differs from S in that P does not single-step through functions and procedures called by the current statement. P treats an entire nested call as a single statement; thus procedure calls and function invocations are completed before program control returns to the Debugger, allowing you to bypass the detailed execution of routines (e.g., ones already debugged). If the current procedure ends, P begins single-stepping the procedure that called the current procedure.

The P(*n*) command is equivalent to repeating the P command *n* times.

As with the C command, you may not go past the end of the program with an S or a P command. If you do so, you receive an error message telling you to use G to restart the program.

First paragraph of handwritten text, starting with a capital letter.

Second paragraph of handwritten text, continuing the narrative.

Third paragraph of handwritten text, showing a change in the subject.

Fourth paragraph of handwritten text, concluding the main body of the letter.

Fifth paragraph of handwritten text, possibly a closing or signature area.

Pascal-2 V2.1/RT-11 Debugger Guide

Examples of the S and P Commands

.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} B(main,9)

} G

1 2 3 4 5 6 7

Left,Right? 1 5 ----- input to ROTAT

Breakpoint at MAIN,9 Rotate(Left, Right, N);

} S

Breakpoint at ROTATE,1 begin;

} S

Breakpoint at ROTATE,2 for I := First to Last do

} S

Breakpoint at ROTATE,3 A[I] := A[I + 1];

} S

Breakpoint at ROTATE,3 A[I] := A[I + 1];

} S(4)

Breakpoint at ROTATE,4 A[Last] := A[First];

} G

Rotated 1 thru 5= 2 3 4 5 2 6 7 ----- further indication of run-time error

} G

1 2 3 4 5 6 7

Left,Right? 1 5 ----- input to ROTAT

Breakpoint at MAIN,9 Rotate(Left, Right, N);

} P

Breakpoint at MAIN,10 for I := 1 to Arraylen do

} P

Breakpoint at MAIN,11 write(N[I]:2);

} P

Breakpoint at MAIN,11 write(N[I]:2);

} P

Breakpoint at MAIN,11 write(N[I]:2);

} P(6)

Rotated 1 thru 5= 2 3 4 5 2 6 7 ----- same indication of a problem

}

THE UNIVERSITY OF CHICAGO

LIBRARY

1911

1911

THE UNIVERSITY OF CHICAGO

LIBRARY

1911

1911

1911

1911

THE UNIVERSITY OF CHICAGO

LIBRARY

1911

1911

1911

1911

1911

THE UNIVERSITY OF CHICAGO

Tracking Commands

Two commands help you track program execution. The **H** command lists the statements that have brought you to your present position. The **T** command traces program execution through each statement.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the **H** command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the **H** command shows the steps leading to the statement causing the error. The **H** command with no parameters prints a list of the last 10 statements executed. **H(n)** prints the last *n* statements up to 50.

The **H** command has other important functions as well. See "Execution Stack Commands" for details and for examples of the command.

T(): Execution Trace

The **T** command accepts a Boolean parameter, either enabling or disabling the tracing of program execution. When tracing is enabled with the **T(TRUE)** command, each statement is identified by its block name and statement number and is displayed before being executed.

A Control-C (^C) interrupts the trace and returns the Debugger to command mode. You may then turn off tracing with the **T(FALSE)** command and continue running your program with the **C** command.

Example of the T Command

.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(main.9.3)

```

27  9      Rotate(Left, Right, N);
28  10     for I := 1 to Arraylen do
29  11         write(N[I]:2);

```

} B(main.9) _____ set breakpoint

} G

1 2 3 4 5 6 7

Left,Right? 1 3 _____ input to ROTAT

Breakpoint at MAIN,9 Rotate(Left, Right, N);

} T(TRUE)

} G

1. The first part of the paper is devoted to a general discussion of the problem.

2. In the second part, we consider the case of a single particle. We show that the motion is periodic and that the period is independent of the energy.

3. In the third part, we consider the case of two particles. We show that the motion is also periodic and that the period is independent of the energy.

4. In the fourth part, we consider the case of three particles. We show that the motion is also periodic and that the period is independent of the energy.

5. In the fifth part, we consider the case of four particles. We show that the motion is also periodic and that the period is independent of the energy.

6. In the sixth part, we consider the case of five particles. We show that the motion is also periodic and that the period is independent of the energy.


```

ROTATE,1  begin _____ tracing output begins
ROTATE,2  for I := First to Last do
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,4  A[Last] := A[First];
ROTATE,5  write('Rotated ',first: 1,' thru ',last: 1,'=');
MAIN,10   for I := 1 to Arraylen do
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,12   writeln
Rotated 1 thru 3= 2 3 2 4 5 6 7 _____ our run-time error is still evident
} I(FALSE) _____ tracing off
} K
} G
  1 2 3 4 5 6 7
Left,Right? 1 3 _____ input to ROTAT
Rotated 1 thru 3= 2 3 2 4 5 6 7
}

```


Data Commands

Debugger data commands allow you to display the current values of variables and to assign new values to them. The data commands provide full access to user identifiers and type definitions. The data commands conform to Pascal type compatibility rules.

W(): Write Variable Value

You use the **W** command to write the value of a variable (including a pointer), of a constant, or of a memory location. The format for the **W** command is:

} W(name1.name2.name3. ...)

where *name* is the name of the variable you want written. As shown, you may write the value of more than one variable by separating variable names with commas.

The type of variable determines the format of the output. For example, integers are displayed as signed decimal numbers. Set variables are displayed in Pascal set notation. Scalar variables are displayed as the names of the enumerated types they represent.

You may use the Pascal colon notation ':' to alter the way variables are written. For example, to print the integer variable *I* as a hexadecimal number, you use:

} W(I:-1)

Also see the example after Variable Assignment.

Real numbers may be formatted according to the same rules used by the compiler.

A numeric constant is used as an address if you wish to write the integer value contained in a memory location. A 'B' placed after the number, as in **W(27740B)**, specifies an octal memory location. Memory locations are displayed as signed integers.

The Debugger may write any complex Pascal data structure, including records and arrays, except files.

The Debugger displays an array in an orderly fashion that reflects the array's structure. For each change in the least significant (rightmost) index of the array, the Debugger writes a space between elements. For each change in the next least significant (second-from-rightmost) index, the Debugger starts a new line. And for changes in the *n*th index, where *n* is the number of "places from the right" of the least significant index and *n* is greater than 2, the Debugger writes *n* - 2 blank lines and indents the first row of the display *n* - 2 spaces.

The first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

Pascal-2 V2.1/RT-11 Debugger Guide

Example:

Pascal-2 RT11 SJ V2.1D 9-Feb-84 7:06 AM Site #1-1 Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
MULTI/DEBUG

```
Line Stmt
1      program Multi;                { multidimensional variables }
2
3      var A: array [1..3, 1..3, 1..3] of integer;
4          I, J, K: integer;
5
6      1 begin
7          2 for I := 1 to 3 do
8              3 for J := 1 to 3 do
9                  4 for K := 1 to 3 do
10                     5 A[I,J,K] := (I * 10 + J) * 10 + K;
11      end.
```

*** No lines with errors detected ***

.RUN MULTI

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program MULTI

```
} G
} W(A)
111 112 113
121 122 123
131 132 133

211 212 213
221 222 223
231 232 233

311 312 313
321 322 323
331 332 333
```

When you write records, the Debugger lists each field name followed by the value of that field. The format of each field is determined by the data type of the field. Complex records, such as those containing arrays of records, can get messy; you may want to have the listing on hand to show the definition of the record being printed.

Variable Assignment

The Debugger command to modify the value of a program variable is identical in form to a Pascal assignment statement. The left-hand side of the '=' assignment operator indicates the variable to be modified. This variable may include array indices, record field selectors, and pointer accesses. The right-hand side specifies the value to be assigned. This may be a simple constant or literal value, or another program variable. Standard notation is used for all values, including sets. General expressions (operators and functions) are not permitted.

Debugger variable assignments must conform to the Pascal assignment compatibility rules. All variables accessed in an assignment command must be available in the current stack context. The E(n) command may be used to temporarily change context, if necessary.

Examples of the W Command and Variable Assignment

Pascal-2 RI11 SJ V2.1D 9-Feb-84 7:06 AM Site #1-1 Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
COLOR/DEBUG

```

Line Stmt
1      program Color;
2
3      type
4      Color = (Red, Orange, Yellow, Blue, Green);
5
6      var
7      c: Color; I: integer;
8      Colorset: set of Color;
9      a: array [0..4] of Color;
10     r: record
11         I: integer;
12         S: set of Color;
13         C: packed array [1..4] of char;
14     end;
15
16     1 begin
17     2   for C := Red to Green do A[ord(C)] := C;
18     4   Colorset := [Red, Yellow..Green];
19     5   R.I := 123; R.S := [Orange, Green]; R.C := 'TEST';
20     end.

```

*** No lines with errors detected ***

.RUN COLOR

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program COLOR

> G

> W(A)

RED ORANGE YELLOW BLUE GREEN

> A[1] := Red; A[4] := Red; W(A)

RED RED YELLOW BLUE RED

Pascal-2 V2.1/RT-11 Debugger Guide

```
} W(Colorset)  
[RED,YELLOW..GREEN]  
} Colorset := [Red..Green]: W(Colorset)  
[RED..GREEN]  
} W(R)  
I: 123  
S: [ORANGE, GREEN]  
C: TEST  
  
} R.I := 321: R.S := Colorset: W(R)  
I: 321  
S: [RED..GREEN]  
C: TEST  
  
}
```


Informational Commands

Informational commands show data being maintained by the Debugger. The D command shows the current breakpoints, user-defined macros, and variables being watched. The L command shows selected parts of the program listing, so that you won't have to reprint the listing each time you revise your program.

D: Display Parameters

The D command displays all breakpoints, user-defined macros, and the variables being watched; it also shows any commands associated with each. Breakpoints are set with the B command. Macros are stored Debugger commands created by the M command and executed by the X command. The V command is used to set variable watches. (See the respective sections for details on these commands.)

See the ROTAT.PAS example in "Running the Debugger" and the example after the C command.

L, L(): List Source Lines

The L command uses the statement numbers in the listing file of your program to list portions of the source program. The L command allows you to list individual statements, parts of procedures, or entire procedures.

When a breakpoint is set at a statement with B(), the Debugger prints only the first line associated with the statement. The History command H also prints only the first line of the statement. The L command, in contrast, prints all lines containing the statement.

The L command with no parameters lists the current procedure. You may list any other procedure by giving the procedure name enclosed in parentheses. For example, L(MAIN) lists the body of the main program.

The command L(*proc*,*stmtnum*) lists a single statement, where *proc* is the name of the procedure and *stmtnum* is the number of the statement to print.

You also may list sections of the program starting or ending at a particular statement by specifying a line count after the statement number. For example, L(MAIN,1,10) lists the first ten lines of the main program.

The general form of the command is:

```
) L(proc,stmtnum,count)
```

where *proc* and *stmtnum* describe a statement in the program. A positive *count* prints that many lines starting at the statement specified and moving forward. A negative *count* causes the Debugger to list statements up to and including *stmtnum*. (The listing of source lines in external functions and procedures requires a slightly different form of the L command. See "Debugging External Modules" later in this guide for details.)

This example lists 5 lines beginning with the first statement of procedure ROTATE:

```
) L(Rotate,1,5)
14 1      begin
15 2      for I := First to Last do
16 3          A[I] := A[I + 1];
17 4      A[Last] := A[First];
18 5      write('Rotated ',first: 1,' thru ',last: 1,'=');
```

1890

Received of the Treasurer of the
Board of Education the sum of \$100.00
for the year 1890.

Witness my hand and seal this 1st day of
January 1891.

Attest:
The Board of Education of the City of New York
do hereby certify that the above is a true and
correct copy of the original record of the
Board of Education of the City of New York
for the year 1890.

Witness my hand and seal this 1st day of
January 1891.

Attest:
The Board of Education of the City of New York
do hereby certify that the above is a true and
correct copy of the original record of the
Board of Education of the City of New York
for the year 1890.

Pascal-2 V2.1/RT-11 Debugger Guide

This example lists 2 lines leading up to and including the 4th statement of procedure Rotate:

```
  } L(Rotate,4,-2)  
    16      3      A[I] := A[I + 1];  
    17      4      A[Last] := A[First];
```

When you list an entire procedure, the Debugger attempts to include the procedure heading and local variable declarations in the listing. However, this header information is only used by the Pascal compiler, so the Debugger has to estimate where the procedure header information is located in the listing file. As a result, the Debugger may not always print the complete header information or may sometimes print part of the preceding procedure.

Long procedures may take some time to print. A single Control-C (^C) interrupts the listing and returns the Debugger to command mode.

Utility Commands

Utility commands allow you to define a series of commands as a macro to be executed by entering a single command.

M(): Define Macro

The **M** command saves you some typing when you need to issue repetitive commands. For example, you may need to write the value of several critical variables at different places in your program. The **M** feature lets you combine these commands under one name, then execute this group of commands by using the **X** command, explained below. You cannot pass parameters to macros.

The format for definition of a macro is:

```
  } M(name)<command1; command2; command3; ...>
```

where *name* is any alphanumeric string containing up to 32 symbols. The **X** command uses *name* to identify the macro. You may place as many Debugger commands in the angle brackets '< >' as fit on one command line. You may delete a macro by typing **M(name)** with no commands. Available memory is the only limit on the number of macros you may define. The **D** command lists macro names and the commands associated with each name.

See the example after the **X** command.

X(): Execute Macro

You may execute the Debugger commands associated with a macro by using the **X** command. The format is:

```
  } X(name)
```

where *name* is the name of the macro. The effect of the **X** command is to execute the Debugger commands defined by the **M** command of that name.

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

Examples of the M and X Commands

.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} M(DumpN)<W('The value of N=',N)> _____ define macro} B(Rotate,1): G

1 2 3 4 5 6 7

Left,Right? 2 6 _____ input to ROTAT

Breakpoint at ROTATE,1 begin

} M(DumpI)<W('I=',I)> _____ define macro

} Q

Breakpoints

ROTATE,1 begin

Macros

DUMPI W('I=',I)

DUMPN W('The value of N=',N)

} S

Breakpoint at ROTATE,2 for I := First to Last do

} S

Breakpoint at ROTATE,3 A[I] := A[I + 1];

} X(DumpI) _____ execute macro

I= 2

} S(4): X(DumpI): X(DumpN)

Breakpoint at ROTATE,3 A[I] := A[I + 1];

I= 6

The value of N= 1 3 4 5 6 3 7

} Q

5. [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Execution Stack Commands

The execution stack commands allow you to trace down run-time errors by examining the stack. The **H** command shows at any time a history of program execution and the current stack of active procedure and function calls. The **I** command lists the names of the parameters and local variables in any procedure in the execution stack. The **E** command allows you to change the context of the stack frame from the current procedure to another so you may access variables you otherwise wouldn't be able to.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the **H** command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the **H** command shows the steps leading to the statement causing the error. The **H** command with no parameters prints a list of the last 10 statements executed. **H(n)** prints the last *n* statements up to 50.

The **H** command also lists the execution stack. Each time a procedure or function is called, a new entry is made at the top of the execution stack. When the procedure exits, that entry is removed from the top of the stack. The main program is always at the bottom of the stack. The **H** command shows the procedures that were called to get from the main program to the current procedure. **H(0)** prints just the execution stack.

In the display, each procedure or function in the execution stack is identified by a number. This procedure number is used to identify procedures in the execution stack for the **I** and **E** commands described in following sections. (These are not the statement numbers used to identify other Debugger commands.)

In the display, the '**<**' character marks the current procedure. Unless the **E** command is used the current procedure is always the top procedure in the execution stack. The Debugger uses the current procedure to determine the local variables that can be accessed according to Pascal scope rules. Procedures marked with the asterisk '*****' character are those procedures that contain the lexical definition of the current procedure. The parameters and local variables in the procedures marked by '**<**' or '*****' are the only local variables that you may look at or change directly. If you wish to look at local variables in other procedures in the execution stack, you must use the **E** command.

See the example after the **E** command.

1917

1917

1917

1917

1917

1917

1917

1917

H, H(): Names of Variables

The **H** command with no parameters lists the names of the parameters and local variables in the current procedure. If you are in the main program, the command displays all of the global-level variable names.

H with a numeric parameter lists the names of the local variables in the procedure so numbered on the execution stack. These numbers are obtained via the **H** command, described above.

Note that **H** lists the names of the local variables and parameters in any procedure or function on the stack, not merely the ones marked with the 's'. However, you cannot write or change the values of variables unless they are in procedures or functions marked with the 's'.

The **E** command allows access to variables that you otherwise cannot access from the current procedure.

See the example after the **E** command.

E(): Enter Stack-Frame Context

The Debugger normally enforces Pascal scope rules. If you stop your program in the middle of a procedure, you may write or modify only those variables and parameters of the procedures that enclose the current procedure, as described in the section on the **H** command.

To look at or change local variables in procedures that are not accessible to the current procedure, the **E** command gets around the Pascal scope rules by temporarily changing the context of the current procedure.

The **H** command numbers the procedures in the execution stack. The main program is always 1, and procedures called from the main program are listed as 2, and so on. If you want to examine variables in procedure 5 in the current execution stack, and it is not marked with an 's' (and therefore not available to you from where you are), you use **E(5)** to temporarily enter the context of that procedure.

The **E** command affects only debugging commands that follow it on the same command line. For example, to print the value of the variable **I** in the procedure listed as 5, you type:

```
> E(5): W(I)
```

This command line makes procedure 5 the current procedure. Then, using the context of procedure 5, the Debugger prints the value of the variable **I**. At the end of the command line, the current procedure is changed back to the top procedure in the execution stack.

Because the **H** command allows you to list the names of variables in all the procedures on the execution stack, the following commands are equivalent:

```
> E(5): H  
> H(5)
```

The following information was obtained from the records of the Department of the Interior, Bureau of Land Management, on the subject of the land in question.

The land in question is located in the County of [County Name], State of [State Name], and is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section.

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

The land in question is situated in the [Township Name] Township, [Range Name] Range, [Section Name] Section, and is situated in the [County Name] County, State of [State Name].

Examples of the H, N, and E Commands

```
Breakpoint at CHECK,1 begin { start of check }
} H(5) _____ list last 5 statements executed
```

Program execution history:

```
ANALYZEMOVE,9 Vacant[Target] := false;
ANALYZEMOVE,10 if CentralSquares[Target] then
ANALYZEMOVE,14 PossibleMoves := PossibleMoves+1;
ANALYZEMOVE,15 Check(4); Check(5); Check(-4); Check(-5);
CHECK,1 begin { start of check }
```

Procedure execution stack

```
8< CHECK,1 begin { start of check }
7* ANALYZEMOVE,15 Check(4); Check(5); Check(-4); Check(-5);
6* ANALYZE,12 AnalyzeMove(4,I); AnalyzeMove(5,I);
5* EVALUATEBOARD,4 Analyze;
4 GENMOVE,15 EvaluateBoard(N*,Turn);
3 MOVEPIECE,9 if MovesAllowed then GenMove(I,J);
2 EXPAND,11 if Color[Who]=Turn then MovePiece(I,I,0,0);
1* MAIN,8 Expand(Root,True);
```

```
} H _____ local names
DIRECTION SRC DST F
} H(7) _____ names in frame 7
DIRECTION I SAFE WASKING TARGET THRT
} H(4) _____ names in frame 4
I J W OLDPIECE
} E(7): W(1) _____ change context to frame 7, write value
14
} E(4): W(1) _____ change context to frame 4, write value
27
} E(4): H(0)
```

Procedure execution stack

```
8 CHECK,1 begin { start of check }
7 ANALYZEMOVE,15 Check(4); Check(5); Check(-4); Check(-5);
6 ANALYZE,12 AnalyzeMove(4,I); AnalyzeMove(5,I);
5 EVALUATEBOARD,4 Analyze;
4< GENMOVE,15 EvaluateBoard(N*,Turn);
3* MOVEPIECE,9 if MovesAllowed then GenMove(I,J);
2* EXPAND,11 if Color[Who]=Turn then MovePiece(I,I,0,0);
1* MAIN,8 Expand(Root,True);

}
```

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

1870

Stepping Through a Debugger Session

You seldom use only a single Debugger command at any one session, so no single-command example can demonstrate the context in which certain commands are used nor can it demonstrate all of the ways in which certain commands relate. Our approach, therefore, is to step through a sample program to demonstrate some of the common commands in a problem/example context.

In previous sections of this guide, several examples of run-time errors occurring in the execution of ROTAT.PAS were demonstrated. Let's begin this debugging session by running the program and taking a closer look at what is going wrong.

```
.R PASCAL
*ROTAT/DEBUG
```

```
.LINK ROTAT.SY:PASCAL
```

```
Pascal Debugger V3.00 -- 29-Nov-1983
```

```
Debugging program ROTAT
```

```
}
```

After the Debugger has taken control of the program, we instruct it to go (G), then we enter the starting point (2,6).

```
} G
1 2 3 4 5 6 7
Left,Right? 2 6
Rotated 2 thru 6= 1 3 4 5 6 3 7
```

The problem we noted earlier persists. Our starting number in the rotation is apparently incremented by 1 each time the program is run. In this case, the second 3 in our rotated sequence should be 2. With the L command, we now list the part of the main program that initializes the N array. From this, we can choose a location for a breakpoint once the array is initialized.

```
} L(main.1.5) _____ list 5 lines of main program
21 1 begin { Main program }
22 2 for I := 1 to Arraylen do
23 3 begin N[I] := I; write(I:2); end;
24 5 writeln; write('Left,Right? ');
25 7 readln(Left, Right);
} B(main.6) _____ set breakpoint at MAIN,6
} G _____ begin execution
1 2 3 4 5 6 7
Breakpoint at MAIN,6 writeln; write('Left,Right? ');
} V(N[6]) _____ write value of N[6]
6
```

(Note the way in which the Debugger counts statements when more than one is placed on a line, as on line number 24 above. Though not explicitly listed, the second statement on line 24 is statement number 6 and must be identified as such.)

Examination of the array N at this breakpoint shows the array to be correct; the change to the value of the variable must be occurring somewhere else. Using the V (watched variable) command, we tell

Pascal-3 V2.1/RT-11 Debugger Guide

the Debugger to stop the program whenever `N[6]` is changed.

```
} V(N[6]) _____ watch for changes of value of N[6]
} G _____ continue execution
Left,Right? 2 6 _____ input to ROTAT
The value of "N[6]" was changed by the statement:
ROTATE,3 A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4 A[Last] := A[First];
}
```

This is an expected change based on the algorithm being used in the rotation. Our last number is first incremented by 1 before being replaced by the first. Therefore, we continue to watch the variable.

```
} G
The value of "N[6]" was changed by the statement:
ROTATE,4 A[Last] := A[First];
Old value: 7
New value: 3
Breakpoint at ROTATE,5 write('Rotated ',first:1,' thru ',last:1,'=');
} V(First,Last) _____ write values of First and Last
2 6
} V(N) _____ write values of array N
1 3 4 5 6 3 7

} Q _____ quit the Debugger
```

At ROTATE,4 the first element is assigned to the last element after the first element has already been changed. We must introduce a temporary variable to hold the first element value so that it will not be destroyed. We correct the program (adding a "temp" variable, an assignment at line 14 and another between lines 16 and 17), then recompile with the /debug switch. Again, we use the L command to inspect the part of the program we changed.

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

```
} L(ROTATE.1.6) _____ list 6 lines of procedure ROTATE
14 1 begin Temp := A[First];
15 3 for I := First to Last do
16 4 A[I] := A[I + 1];
17 5 A[Last] := Temp;
18 6 write('Rotated ',first: 1,' thru ',last: 1,'=');
19 end;

} G _____ begin execution
1 2 3 4 5 6 7
Left,Right? 2 6 _____ input to ROTAT
Rotated 2 thru 6= 1 3 4 5 6 2 7

} G _____ begin execution
1 2 3 4 5 6 7
Left,Right? 3 4 _____ input to ROTAT
Rotated 3 thru 4= 1 2 4 3 5 6 7
Breakpoint at MAIN,12 writeln
```

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that this is crucial for ensuring the integrity of the financial system and for providing a clear audit trail. The text also mentions the need for regular reviews and updates to the records to reflect any changes in the data.

In addition, the document highlights the role of technology in streamlining the record-keeping process. It suggests that using specialized software can help reduce the risk of human error and make it easier to manage large volumes of data. The text also notes that technology can facilitate the sharing of information between different departments and organizations.

Furthermore, the document discusses the importance of training and education for the personnel responsible for maintaining the records. It states that staff should be kept up-to-date on the latest best practices and technologies to ensure that the records are maintained to the highest standards. The text also mentions the need for ongoing monitoring and evaluation of the record-keeping process to identify any areas for improvement.

Overall, the document stresses that maintaining accurate records is a fundamental responsibility for any organization. It provides a comprehensive overview of the key principles and practices that should be followed to ensure the reliability and accuracy of the financial records. The text also offers practical advice on how to implement these principles in a real-world context.

The second part of the document focuses on the specific steps that should be taken to implement a robust record-keeping system. It begins by outlining the need for a clear policy and procedure that defines the roles and responsibilities of all involved parties. The text then discusses the importance of establishing a consistent naming convention for all records to ensure that they can be easily identified and located.

Next, the document describes the process of creating and maintaining a master index of all records. It explains that this index should be updated regularly to reflect any new records that are added to the system. The text also mentions the need for a backup and recovery plan to protect the records in case of a disaster. Finally, the document discusses the importance of regular audits to verify the accuracy and completeness of the records.

In conclusion, the document provides a detailed guide to implementing a successful record-keeping system. It covers all the essential aspects, from policy development to ongoing monitoring and evaluation. By following the principles and practices outlined in the document, organizations can ensure that their financial records are accurate, reliable, and easy to manage. The text also offers a range of practical tips and suggestions to help organizations overcome common challenges in the record-keeping process.

The document is a comprehensive resource for anyone responsible for maintaining financial records. It provides a clear and concise overview of the key principles and practices that should be followed to ensure the reliability and accuracy of the records. The text is written in a professional and authoritative tone, and it includes a range of practical advice and suggestions to help organizations implement a robust record-keeping system. The document is a valuable tool for anyone looking to improve their record-keeping practices and ensure the integrity of their financial system.

Now the program seems to be running correctly, but let's make one more test before we've satisfied ourselves that the program is running as we want. Note that the G command restarts the program even after it has terminated.

```

} G _____ begin execution
  1 2 3 4 5 6 7
Left,Right? 1 7 _____ input to ROTAT

II2 -- Fatal error at user PC=1424
Array subscript out of bounds

}

```

The end points of a data subrange are always good places to look for run-time errors such as "Array subscript out of bounds," because they are the values most likely to exceed the predefined limits. We begin analyzing this new error by writing the values of variables found in the line where the error occurred.

```

} W(I) _____ write the value of I
7
} W(A[8]) _____ write the value of A[8]
Array subscript too large
W(A[8])

The limits are 1..7
} Q _____ quit the Debugger

```

Now we can diagnose the error. We see that the limits for the array subscript of A have been exceeded by one. The for loop in the ROTATE procedure is likely to be looping too many times. We reduce the final value by 1 (last becomes last-1 in line 15) and recompile the program. When we run the program, we tell the Debugger to list procedure ROTATE, so that we can more closely follow the section of the program we changed.

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

```

} L(ROTATE.1.6) _____ list 6 lines of procedure ROTATE
  14  1      begin Temp :=A[First];
  15  3      for I := First to Last - 1 do
  16  4          A[I] := A[I + 1];
  17  5          A[Last] := Temp;
  18  6      write('Rotated ',first: 1,' thru ',last: 1,'=');

} G _____ begin execution
  1 2 3 4 5 6 7
Left,Right? 1 7 _____ input to ROTAT
Rotated 1 thru 7= 2 3 4 5 6 7 1

} Q _____ quit the Debugger

```

The results our program gives are now exactly what they should be. Once satisfied that the program is correct, we recompile it without the debug switch to reduce file and memory requirements, to improve execution speed, and to reinstate the walkback.

Handwritten header or title at the top of the page.

First main paragraph of handwritten text.

Second main paragraph of handwritten text.

Third main paragraph of handwritten text.

Fourth main paragraph of handwritten text.

Fifth main paragraph of handwritten text.

Handwritten footer or concluding text at the bottom of the page.

Debugging External Modules

An external module consists of one or more Pascal procedures or functions written and compiled independently of the main program that invokes it. The Debugger's ability to debug external modules allows you to fully debug an entire program, including externals, and also allows you to debug external procedures and functions only, in the context of a main program.

The debugging of external procedures and functions is simply a matter of compiling the module with the `debug` and `nomain` switches, linking the module with the main program, and upon execution, supplying the module name on certain Debugger commands (see below). The `debug` compilation creates the necessary symbol table files for the module. (Remember to compile the main program with the `debug` switch, too.) Refer to "External Modules" in the Programmer's Guide for rules on the use of external modules.

As mentioned earlier, external modules cannot be debugged directly; they must be called from a main program. To debug an external procedure or function itself, create a short main program that simply invokes the procedure, then exits. Be sure to initialize any variables required of the call. Then compile the main program using the `debug` switch and task build it with the external module.

Differences in the Commands

This section covers only the differences in command syntax and usage; unless otherwise noted, the Debugger commands work as described in previous sections.

In general, the major differences are:

- The `B`, `K` and `L` commands accept the module name followed by a colon (`:`) as the first argument. These commands allow you to set and kill breakpoints and list source lines in external procedures and functions. The revised syntax for these commands are as follows:

```
} B(module: block, stmtnum) < ... >  
} K(module: block, stmtnum)  
} L(module: block, stmtnum, count)
```

The module name *module* is the name of the source file minus extension. For example, `TEST` is the module name for `TEST.PAS`. *Block* is the name of the procedure or function being referenced in *module*. The other arguments are the same as described in earlier sections.

- When displaying breakpoint and source-line information, the Debugger includes the module name along with the procedure name and line number. The `D` command, in addition to displaying breakpoints, user-defined macros, and variables being watched, shows you which module is currently being debugged.
- Defaults apply to the current module being debugged. To list lines, set breakpoints or kill breakpoints in any module but the current one, you must specify at least the module name and the procedure name on the commands.
- If you try to list lines or set/kill breakpoints in an external module **not** compiled for debugging, the run-time error "can't open file" causes the Debugger to abort trying to open the listing and symbol table files for that module. Of course, if you have old listing and symbol table files for that module lying around, the Debugger opens these files even if you did not wish to debug that module. In this case, if the data files do not match the source you're using, your results may not be accurate.

An example is presented in the next section.

1. The first part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

2. The second part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

3. The third part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

4. The fourth part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

5. The fifth part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

6. The sixth part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

7. The seventh part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

8. The eighth part of the document is a letter from the President of the United States to the Congress, dated January 3, 1863. It is a very important document, as it contains the President's message to Congress, and is one of the most important documents in the history of the United States.

Example

To illustrate the debugging of external modules, we present a single debugging session in which the above commands are used. In this example, the main program ROTAT.PAS, presented earlier, calls the external procedure Rotate contained in XROT.PAS. (In the previous program, procedure Rotate was a local procedure.)

After compiling the main program and external modules and task building them, run the program.

```
.R PASCAL
*ROTAT/DEBUG
```

```
.LINK ROTAT,ROTATE,SY:PASCAL
.RUN ROTAT
```

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

```
} L _____ defaults to main program
14 1 begin { main program }
15 2   for I := 1 to Arraylen do
16 3     begin N[I] := I; write(I: 2); end;
17 5   writeln; write('Left,Right? ');
18 7   readln(Left, Right);
19 8   I := 4;
20 9   Rotate(Left, Right, N);
21 10  for I := 1 to Arraylen do
22 11    write(N[I]: 2);
23 12  writeln
24    end.
} B(MAIN,7)
} B(XROT:ROTATE,5)<W(TEMP)>
```

Initially, the L command without parameters lists the main program by default because it is in the current module being debugged. The first breakpoint command could just as easily be B(ROTAT:MAIN,7). However, the module name is not necessary because the main program is the current module. The second breakpoint command shows that you must supply the module name for modules other than the current one. With the G command, program execution begins:

```
} G
1 2 3 4 5 6 7
Breakpoint at ROTAT:MAIN,7 readln(Left, Right);
} G
Left,Right? 1 7 _____ input to ROTAT
Breakpoint at XROT:ROTATE,5 A[Last] := Temp;
1
```

Handwritten header or title at the top of the page.

First paragraph of handwritten text, starting with a capital letter.

Second paragraph of handwritten text, continuing the narrative.

Third paragraph of handwritten text, possibly containing a list or detailed notes.

Fourth paragraph of handwritten text, appearing as a separate section.

Fifth paragraph of handwritten text, located near the bottom of the page.

Final line of handwritten text at the bottom of the page.

Pascal-2 V2.1/RT-11 Debugger Guide

Note the way the Debugger reports the breakpoints. At this point the current procedure being debugged is the procedure `Rotate`, in external module `XROT`. (The single '1' is the value of `Temp` when the breakpoint is reached.) Now the `B`, `K` and `L` commands default to the external procedure `Rotate`, as shown below for the `L` command. To list the lines in the main program, you must specify the module name, as shown in the second `L` command:

```
> L
16 1 begin Temp := A[First];
17 3   for I := First to Last-1 do
18 4     A[I] := A[I + 1];
19 5   A[Last] := Temp;
20 6   write('Rotated ', first: 1, ' thru ', last: 1, '=');
21   end;
```

```
> L(ROTAT:MAIN)
14 1 begin { main program }
15 2   for I := 1 to Arraylen do
16 3     begin N[I] := I; write(I: 2); end;
17 5   writeln; write('Left,Right? ');
18 7   readln(Left, Right);
19 8   I := 4;
20 9   Rotate(Left, Right, N);
21 10  for I := 1 to Arraylen do
22 11    write(N[I]: 2);
23 12  writeln
24   end.
```

```
> D _____ display current module and breakpoints
Current module: XROT
```

Breakpoints

```
XROT:ROTATE,5 A[Last] := Temp;
               <W(TEMP)>
```

```
ROTAT:MAIN,7 readln(Left, Right);
```

```
> H _____ History command
```

Program execution history:

```
XROT:ROTATE,1 begin Temp := A[First];
XROT:ROTATE,2 begin Temp := A[First];
XROT:ROTATE,3 for I := First to Last-1 do
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,4 A[I] := A[I + 1];
XROT:ROTATE,5 A[Last] := Temp;
```

Procedure execution stack

```
3< XROT:ROTATE,5 A[Last] := Temp;
2< Module: XROT
1< ROTAT:MAIN,9 Rotate(Left, Right, N);
```

On the 1st day of the month of January 1900, the undersigned, being duly sworn, depose and say that the within and foregoing is a true and correct copy of the original of the same, as the same appears from the records of the Court of the County of [County Name] State of [State Name], and that the same is a true and correct copy of the original of the same, as the same appears from the records of the Court of the County of [County Name] State of [State Name].

Subscribed and sworn to before me on the 1st day of the month of January 1900, at the County of [County Name] State of [State Name].

Notary Public for the State of [State Name]

My Commission Expires on the [Date] day of [Month] 190[Year]

Witness my hand and the seal of my office this 1st day of January 1900.

Notary Public for the State of [State Name]

My Commission Expires on the [Date] day of [Month] 190[Year]

Witness my hand and the seal of my office this 1st day of January 1900.

Notary Public for the State of [State Name]

Debugging External Modules

Execution continues with the C command. The K and B commands used below demonstrate the rules governing the setting and removing of breakpoints. Note the erroneous breakpoint command and the corrected command.

```
} C _____ continue execution  
Rotated 1 thru 7= 2 3 4 5 6 7 1
```

Program terminated.

Breakpoint at ROTAT:MAIN,12 writeln

```
} K(MAIN,7) _____ kill breakpoint
```

```
} C
```

```
1 2 3 4 5 6 7
```

```
Left,Right? 1 7 _____ input to ROTAT
```

Breakpoint at XROT:ROTATE,5 A[Last] := Temp;

```
1
```

```
} B(MAIN,7) _____ won't work without the module name
```

No such statement in this procedure

```
B(MAIN,7)
```

```
} B(ROTAT:MAIN,7) _____ that's better
```

```
} C
```

```
Rotated 1 thru 7= 2 3 4 5 6 7 1
```

```
} Q _____ quit the Debugger
```

1. The first part of the document

2. The second part of the document

3. The third part of the document

4. The fourth part of the document

5. The fifth part of the document

6. The sixth part of the document

7. The seventh part of the document

8. The eighth part of the document

9. The ninth part of the document

10. The tenth part of the document

11. The eleventh part of the document

12. The twelfth part of the document

13. The thirteenth part of the document

14. The fourteenth part of the document

15. The fifteenth part of the document

16. The sixteenth part of the document

17. The seventeenth part of the document

18. The eighteenth part of the document

19. The nineteenth part of the document

Overlays

The debugging of large programs is aided by the execution of two command files, EXTRAC.COM and either XMDBG.COM (for XM systems) or SJDBG.COM (for SJ systems), which are included in the distribution kit. These command files produce a much smaller load image than that of a non-overlaid Debugger, saving up to 6K words of memory. These command files can be executed with the indirect (at-sign) processor.

The command file EXTRAC.COM extracts from the Pascal support library the modules needed for overlaying the Debugger against a Pascal program. EXTRAC.COM need only be executed once, at installation, to ensure that the necessary Debugger and support library modules are available. The modules are placed on the system device, with the extension .DBG.

The command file XMDBG.COM, used in the XM environment, links your program and the Debugger into virtual overlays. The command file SJDBG.COM, for the SJ environment, links your program and the Debugger into overlays.

After the modules have been extracted, examine and modify the link command file you wish to use, replacing all occurrences of F00 in the file with the name of your program. Then execute the link command file to create the overlaid program. The program is now ready to run.

When debugging programs that use overlays don't confuse the program's existing overlay structure with the structure imposed by the link command file you are using. The results may be unpredictable.

Appendix A: Debugger Command Summary

B	Remove current breakpoint
B(block,stmtnum)	Set a control breakpoint
B(block,stmtnum)< ... >	Control breakpoint with stored commands
PDP B(module:block,stmtnum)	Set a control breakpoint in external module
B(module:block,stmtnum)< ... >	Set external control breakpoint with stored commands
C	Continue program execution
C(n)	Continue <i>n</i> times
D	Display breakpoints and macros
E(n)	Enter context of frame <i>n</i> (1 line only)
G	Restart program
H	Display recent history and full stack
H(n)	Display last <i>n</i> statements executed
K	Remove all control breakpoints
K(block,stmtnum)	Remove specified breakpoint
K(module:block,stmtnum)	Remove specified breakpoint from external module
L(proc)	List source of <i>proc</i>
L(proc,stmtnum)	List statement <i>stmtnum</i> in <i>proc</i>
L(proc,stmtnum,x)	List <i>x</i> lines beginning with statement <i>stmtnum</i> in <i>proc</i>
L(module:proc)	List source of external module <i>proc</i>
L(module:proc,stmtnum)	List statement <i>stmtnum</i> in external module <i>proc</i>
L(module:proc,stmtnum,x)	List <i>x</i> lines beginning with statement <i>stmtnum</i> in external <i>proc</i>
M(name)<commands>	Define stored command macro
N	List variable names for current frame
N(n)	List variable names for frame <i>n</i>
P	Proceed 1 statement at current level
P(n)	Proceed <i>n</i> statements
Q	Quit Debugger
S	Single-step statement
S(n)	Single-step <i>n</i> statements
T(TRUE/FALSE)	Enable/disable tracing
V(variable)	Set data breakpoint
V(variable)< ... >	Data breakpoint with stored commands
W()	Write list of values
X(name)	Execute named macro command
variable := value	Assign <i>value</i> to <i>variable</i>
?	Help (display command summary)
^C (Control-C)	Immediate breakpoint
^Z (Control-Z)	Exit from Debugger

Published weekly, except during the summer months, when it is published bi-weekly. The subscription price is \$5.00 per annum in advance. Single copies are sold at 15 cents. The journal is published by the American Medical Association, 535 North Dearborn Street, Chicago, Ill. 60610. Second-class postage paid at Chicago, Ill., and at additional mailing offices. Postmaster: Send address changes in this journal to THE JOURNAL OF THE AMERICAN MEDICAL ASSOCIATION, 535 North Dearborn Street, Chicago, Ill. 60610. Acceptance for mailing at special rate of postage provided for in Section 1103, Act of October 3, 1917, authorized on July 1, 1968. Postage paid at Chicago, Ill., and at additional mailing offices. POSTAGE WILL BE PAID BY ADDRESSEE. POSTMASTER: Send address changes in this journal to THE JOURNAL OF THE AMERICAN MEDICAL ASSOCIATION, 535 North Dearborn Street, Chicago, Ill. 60610.

The Pascal-2 Profiler

The Pascal-2 Profiler can help you tune Pascal programs by detecting bottlenecks: small sections of code in which your program spends a disproportionately large amount of time. The Profiler counts the number of times each Pascal statement in your program is executed then prints a summary describing how many times each procedure is called and what percentage of the total statements executed are found in that procedure.

To use the Profiler, you should compile your program with the `profile` switch. (See the Programmer's Guide for details on compilation switches.) The `profile` switch causes the Pascal-2 compiler to generate several auxiliary files. These files, which permit the Profiler to locate the statements and procedures in your program, are the same ones generated by the `debug` switch.

The compilation and steps are shown below, using a program, `CHECKR.PAS`, which plays a game of checkers.

```
.R PASCAL  
*CHECKR/PROFILE  
  
.LINK CHECKR.SY:PASCAL
```

Upon execution, the Profiler takes control of the program and opens the program's auxiliary files created by the Pascal compiler. For large programs, there may be a short pause while the Profiler scans the auxiliary files to build internal data structures.

Next, the Profiler prompts for the name of the profile output file. If you specify a disk file, the default file extension is `.PRO`. Writing a profile to the terminal is practical only for a short program.

Compile and link the program as previously shown, then run it.

```
.RUN CHECKR  
  
profile V2.1B -- 6-Feb-1983  
  
Profiling module: CHECKR  
  
Profile output file name? CHECKR _____ Output goes to CHECKR.PRO  
  
Welcome to CHECKERS _____ Program continues, slowly
```

The Profiler counts the number of times each statement is encountered. This counting of each statement slows down program execution. For this reason, it may not always be possible to profile programs that operate in a time-critical environment.

The Profiler generates a performance outline when the program terminates. Termination occurs when your program reaches the logical end of the program or when the program detects a fatal error condition. A Control-C (^C) interrupts the program and generates a profile at that point. Entering Control-C (^C)'s twice aborts the generation of the profile.

The Profiler listing has the same two columns of numbers as the Debugger listing (one column numbers each line of the source program and the other gives the statement number of the first statement on each line), plus an extra column of numbers at the far left of the listing.

This leftmost column lists the number of times the statement on that line is executed. If more than one statement appears on the line, the count applies only to the first statement on the line. To obtain an accurate count of each statement in the program, you can run your source program through the PASMAT formatter supplied with Pascal-2. The PASMAT 'S' directive reformats the code so that no more than one statement appears on each line. (PASMAT is described in the Utilities Guide.)

If no number is printed in the leftmost column, then that particular statement was never executed. You can sometimes detect logic errors in your program by scanning the profile output to find sections of code or perhaps entire procedures that are never executed.

A summary of the program's execution, procedure by procedure, appears at the end of the profile listing. Procedures are listed in the order they appear in your source code. Three columns of information are displayed for each procedure, as follows:

Statements	This column lists the number of statements that appear in the definition of the procedure.
Times Called	This column shows how many times each procedure is called during program execution.
Statements Executed	This column has two figures. The first is the number of statements executed in the procedure. For example, a procedure that contains 10 assignment statements and is called 5 times will show 50 statements executed in the statements executed column. This direct relationship is valid only for very simple procedures. In most procedures and functions, loops and other control structures cause the number of "statements executed" to be much larger (or smaller) than you may expect at first glance. The second figure in this column is the percentage of statements executed in this procedure as compared to the total number of statements executed in the program. The total number of procedures and statements and the total number of statements executed are printed at the bottom of the procedure execution summary.

The following example profile from CHECKR shows that 2.6 million statements were executed. (To save space, only the Procedure Execution Summary and relevant portions of the profile listing are shown here.) The Profiler listing shows that the program spent most of its time in only a few procedures. For example, the summary shows that 21 percent of the total statements executed were in the 15-statement procedure **Check**. However, **Check** was called 71,212 times, so that percentage does not seem too far out of line. More interesting is that almost half a million statements (17.63 percent) were executed in the procedure **Initialize**. This number seems excessive because the procedure does nothing more than initialize variables and tables each time a board position is analyzed and was only called 1348 times. We may have a problem here.

PROCEDURE EXECUTION SUMMARY

Procedure name	statements	times called	statements	executed
NEWNODE	15	1390	18070	0.69%
INITIALIZE	17	1348	459668	17.63%
SCAN	32	1348	120580	4.62%
CHECK	15	71212	567111	21.75%
ANALYZEMOVE	40	25382	516325	19.80%
ANALYZE	38	1348	298566	11.45%
UNPACKNODE	54	1348	60860	2.33%
PACKNODE	23	1348	22768	0.87%
SCOREGRADIENT	15	1348	250028	9.59%
SCOREBOARD	54	1348	99228	3.80%
EVALUATEBOARD	5	1348	6740	0.26%
DISPLAYBOARD	22	41	5453	0.21%
EXTRACT	18	715	7328	0.28%
KILL	11	1388	13621	0.52%
PRUNE	3	219	657	0.03%
INIT	165	1	1575	0.06%
COMPARE	14	4128	24768	0.95%
INSERT	26	1843	40490	1.55%
DUMPNODE	11	0	0	0.00%
GENMOVE	18	1273	17822	0.68%
GENJUMP	53	75	4389	0.17%
MOVEPIECE	12	1790	32100	1.23%
EXPAND	17	239	20372	0.78%
POSITIONCURSOR	2	0	0	0.00%
MAKEMOVE	55	306	7371	0.28%
DESCEND	26	197	3592	0.14%
FULLEXPAND	45	127	6046	0.23%
READMOVE	6	2	12	0.00%
DECODE	12	0	0	0.00%
READFILENAME	9	0	0	0.00%
GETUSERMOVE	108	1	90	0.00%
MAIN	91	1	2406	0.09%

There are 1032 statements in 32 procedures in this program.
2607836 statements were executed during the profile.

Because we suspect a problem in the procedure Initialize, we examine the profile output associated with that procedure. The first column of numbers is the statement execution count. The second column is the line number of the statement in the source file. The third column of numbers is the statement number of the statement. (This statement number is the same number used by the Debugger.)

Table of Contents

Chapter I	1	1	1
Chapter II	2	2	2
Chapter III	3	3	3
Chapter IV	4	4	4
Chapter V	5	5	5
Chapter VI	6	6	6
Chapter VII	7	7	7
Chapter VIII	8	8	8
Chapter IX	9	9	9
Chapter X	10	10	10
Chapter XI	11	11	11
Chapter XII	12	12	12
Chapter XIII	13	13	13
Chapter XIV	14	14	14
Chapter XV	15	15	15
Chapter XVI	16	16	16
Chapter XVII	17	17	17
Chapter XVIII	18	18	18
Chapter XIX	19	19	19
Chapter XX	20	20	20
Chapter XXI	21	21	21
Chapter XXII	22	22	22
Chapter XXIII	23	23	23
Chapter XXIV	24	24	24
Chapter XXV	25	25	25
Chapter XXVI	26	26	26
Chapter XXVII	27	27	27
Chapter XXVIII	28	28	28
Chapter XXIX	29	29	29
Chapter XXX	30	30	30
Chapter XXXI	31	31	31
Chapter XXXII	32	32	32
Chapter XXXIII	33	33	33
Chapter XXXIV	34	34	34
Chapter XXXV	35	35	35
Chapter XXXVI	36	36	36
Chapter XXXVII	37	37	37
Chapter XXXVIII	38	38	38
Chapter XXXIX	39	39	39
Chapter XL	40	40	40
Chapter XLI	41	41	41
Chapter XLII	42	42	42
Chapter XLIII	43	43	43
Chapter XLIV	44	44	44
Chapter XLV	45	45	45
Chapter XLVI	46	46	46
Chapter XLVII	47	47	47
Chapter XLVIII	48	48	48
Chapter XLIX	49	49	49
Chapter L	50	50	50
Chapter LI	51	51	51
Chapter LII	52	52	52
Chapter LIII	53	53	53
Chapter LIV	54	54	54
Chapter LV	55	55	55
Chapter LVI	56	56	56
Chapter LVII	57	57	57
Chapter LVIII	58	58	58
Chapter LIX	59	59	59
Chapter LX	60	60	60
Chapter LXI	61	61	61
Chapter LXII	62	62	62
Chapter LXIII	63	63	63
Chapter LXIV	64	64	64
Chapter LXV	65	65	65
Chapter LXVI	66	66	66
Chapter LXVII	67	67	67
Chapter LXVIII	68	68	68
Chapter LXIX	69	69	69
Chapter LXX	70	70	70
Chapter LXXI	71	71	71
Chapter LXXII	72	72	72
Chapter LXXIII	73	73	73
Chapter LXXIV	74	74	74
Chapter LXXV	75	75	75
Chapter LXXVI	76	76	76
Chapter LXXVII	77	77	77
Chapter LXXVIII	78	78	78
Chapter LXXIX	79	79	79
Chapter LXXX	80	80	80
Chapter LXXXI	81	81	81
Chapter LXXXII	82	82	82
Chapter LXXXIII	83	83	83
Chapter LXXXIV	84	84	84
Chapter LXXXV	85	85	85
Chapter LXXXVI	86	86	86
Chapter LXXXVII	87	87	87
Chapter LXXXVIII	88	88	88
Chapter LXXXIX	89	89	89
Chapter LXXXX	90	90	90
Chapter LXXXXI	91	91	91
Chapter LXXXXII	92	92	92
Chapter LXXXXIII	93	93	93
Chapter LXXXXIV	94	94	94
Chapter LXXXXV	95	95	95
Chapter LXXXXVI	96	96	96
Chapter LXXXXVII	97	97	97
Chapter LXXXXVIII	98	98	98
Chapter LXXXXIX	99	99	99
Chapter LXXXXX	100	100	100

THESE THINGS ARE THE PROPERTY OF THE UNIVERSITY OF CHICAGO
AND ARE NOT TO BE LOANED OR REPRODUCED IN ANY MANNER
WITHOUT THE PERMISSION OF THE LIBRARY OF THE UNIVERSITY OF CHICAGO

The Profiler listing for procedure Initialize is:

	173		procedure Initialize;
	174		var
	175		I: integer;
1348	176	1	begin { start of Initialize }
1348	177	2	for I := - 5 to 49 do begin
74140	178	3	Vacant[I] := false;
74140	179	4	Friend[I] := false;
74140	180	5	Enemy[I] := false;
74140	181	6	FriendKing[I] := false;
74140	182	7	EnemyKing[I] := false;
74140	183	8	Protected[I] := false;
	184		end;
1348	185	9	Pinned := 0;
1348	186	10	Threatened := 0;
1348	187	11	Umobil := 0;
1348	188	12	Denied := 0;
1348	189	13	BlackPieces := 0;
1348	190	14	WhitePieces := 0;
1348	191	15	Center := 0;
1348	192	16	MoveSystem := 0;
1348	193	17	EnemyHasKings := false;
	194		end; { of Initialize }

In statements 3 through 8, a for loop is initializing several Boolean arrays of the same type. Each assignment inside the loop is executed 74,140 times—a very inefficient way to initialize these arrays. Instead, we can modify the program to initialize one array, then assign that array to the other arrays to be initialized.

The effect of the modification is apparent in this new profile of the same section of code.

	173		procedure Initialize;
	174		var
	175		I: integer;
1732	176	1	begin { start of Initialize }
1732	177	2	for I := - 5 to 49 do begin
95260	178	3	Vacant[I] := false;
	179		end;
1732	180	4	Friend := Vacant;
1732	181	5	Enemy := Vacant;
1732	182	6	FriendKing := Vacant;
1732	183	7	EnemyKing := Vacant;
1732	184	8	Protected := Vacant;
1732	185	9	Pinned := 0;
1732	186	10	Threatened := 0;
1732	187	11	Umobil := 0;
1732	188	12	Denied := 0;
1732	189	13	BlackPieces := 0;
1732	190	14	WhitePieces := 0;
1732	191	15	Center := 0;
1732	192	16	MoveSystem := 0;
1732	193	17	EnemyHasKings := false;
	194		end; { of Initialize }

July 1911		Total	
July 1st	100	100	100
July 2nd	100	200	200
July 3rd	100	300	300
July 4th	100	400	400
July 5th	100	500	500
July 6th	100	600	600
July 7th	100	700	700
July 8th	100	800	800
July 9th	100	900	900
July 10th	100	1000	1000
July 11th	100	1100	1100
July 12th	100	1200	1200
July 13th	100	1300	1300
July 14th	100	1400	1400
July 15th	100	1500	1500
July 16th	100	1600	1600
July 17th	100	1700	1700
July 18th	100	1800	1800
July 19th	100	1900	1900
July 20th	100	2000	2000
July 21st	100	2100	2100
July 22nd	100	2200	2200
July 23rd	100	2300	2300
July 24th	100	2400	2400
July 25th	100	2500	2500
July 26th	100	2600	2600
July 27th	100	2700	2700
July 28th	100	2800	2800
July 29th	100	2900	2900
July 30th	100	3000	3000

The following table shows the total number of cases of smallpox reported in the United States for each month from July 1911 to July 1912. The total number of cases for the entire period is 10,000.

August 1911		Total	
August 1st	100	100	100
August 2nd	100	200	200
August 3rd	100	300	300
August 4th	100	400	400
August 5th	100	500	500
August 6th	100	600	600
August 7th	100	700	700
August 8th	100	800	800
August 9th	100	900	900
August 10th	100	1000	1000
August 11th	100	1100	1100
August 12th	100	1200	1200
August 13th	100	1300	1300
August 14th	100	1400	1400
August 15th	100	1500	1500
August 16th	100	1600	1600
August 17th	100	1700	1700
August 18th	100	1800	1800
August 19th	100	1900	1900
August 20th	100	2000	2000
August 21st	100	2100	2100
August 22nd	100	2200	2200
August 23rd	100	2300	2300
August 24th	100	2400	2400
August 25th	100	2500	2500
August 26th	100	2600	2600
August 27th	100	2700	2700
August 28th	100	2800	2800
August 29th	100	2900	2900
August 30th	100	3000	3000

The Pascal-2 Profiler

The result is clear: Instead of six assignments, each of which is executed 74,140 times, we have one assignment executed 95,260 times. (The execution numbers differ from the sample execution summary because the CHECKR program uses random numbers to play a different game each time it is run.) Overall, the Program Execution Summary shows that the time spent in the Initialize procedure has dropped from 17 percent to 4 percent of the total program. By rewriting six lines, we have improved performance by 11 percent.

Further, the number of times Statement 3 is executed can be reduced by the use of a global array initialized only once at the start of the program.

Similar optimizing techniques may be applied to other parts of the program. The Procedure Execution Summary indicates where the effort can best be applied—and where it cannot. For example, the program spent 21 percent of its time in the 15-statement procedure called *Check*. The trimming of even one statement from this procedure could significantly improve performance. On the other hand, one of the larger procedures in the CHECKR program is *GenJump*, containing 53 statements. The program, however, spent much less than 1 percent of its time in this procedure. Even by eliminating this procedure completely, we would improve program performance by only a trifling amount.

Two warnings: First, a statement count is not identical to "work." Complex statements take more time to execute than simple statements and this time is not measured. Second, the percentages shown in the *statements executed* column are percentages of execution counts, not execution time. For compute-bound programs such as CHECKR, the execution percentage closely approximates the percentage of time spent in the procedures. I/O-bound programs, however, may spend much of their execution time opening files or waiting for the disk to transfer information to memory. In this case, the execution count percentages may differ significantly from the real amount of time spent in the procedures.

Pascal-2 V2.1/RT-11 Utilities Guide

Introduction to the Utilities Guide

The Pascal-2 utilities are a collection of programs designed to make life easier for programmers. Some of the utilities, such as the formatters, are designed to lessen the tedium in formatting programs and program documentation. Other utilities, such as the cross-reference programs, can help analyze code. Still other utilities, such as the MACRO package or the string-processing package, extend the capabilities of Pascal-2.

Each section of the Utilities Guide describes the particular utility in detail and includes examples on its use. Briefly, however, the Utilities Guide contains the following:

Two Program Formatters: PASMAT, a sophisticated formatter with a number of options; PB, a simple formatter designed to assist, rather than supplant, your own formatting of program text.

Two Cross-Reference Programs: XREF, which cross-references the variables in your program or words in a text file; and PROCREF, which cross-references the procedures in your program.

Dynamic String Package: STRING.PAS, a set of procedures designed to help you manipulate character strings.

MACRO Package: PASMAL, which helps to interface MACRO-11 routines with Pascal-2 programs.

Text Formatter: PROSE, which provides a number of formatting options for the production of computer-related documentation.

PASMAT: A Pascal-2 Formatter

PASMAT generates a standard format for Pascal code. PASMAT will accept standard Pascal and the language extensions in Pascal-2. PASMAT accepts full programs, external procedures, or groups of statements. A syntactically incorrect program will cause PASMAT to abort and to cease formatting the output file.

PASMAT's default formatting requires no control from you. The best way to find out how the formatting works is to try it and see. In addition, PASMAT's formatting directives give you considerable control over the output format when you wish.

Overview of Capabilities

PASMAT has these capabilities:

- The program may be converted to uniform case conventions, under the control of the user.
- The program is indented to show its logical structure and to fit into a specified output line length.
- Comment delimiters are changed to braces (`{ }`).
- If requested, the break character (`_`) will be removed from identifiers for use at installations that do not support the break character.
- If requested, the first instance of each identifier will determine the appearance of all subsequent instances of the identifier.
- All non-printing characters are removed; this feature is useful after certain editing bugs.

PASMAT handles comments, statements, and tables in the following manner:

Comments

PASMAT's rules allow you to achieve almost any effect needed in the display of comments.

- A comment standing alone on a line will be left-justified to the current indention level, so that it will be aligned with the statements before and after it. If it is too long to fit with this alignment, it will be right-justified.
- A comment that begins a line and continues to another line will be passed to the output unaltered, indention unchanged. This type of comment is assumed to contain text formatted by the author, so it is not formatted.
- If a comment covered by one of the above rules will not fit within the defined output line length, the output line will be extended as necessary to accommodate the comment. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence.
- A comment embedded within a line will be formatted with the rest of the code on that line. Breaks between words within a comment may be changed to achieve proper formatting, so nothing that has a fixed format should be used in such a comment. If a comment cannot be properly spaced so that the line will fit within the output length, that line will be extended as necessary. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence. If no code follows a comment in the input line, then no code will be placed after the comment in the output line.

THE HISTORY OF THE

... of the ...

... of the ...

... of the ...

... of the ...

... of the ...

... of the ...

... of the ...

... of the ...

... of the ...

Statement Bunching

The normal formatting rule for a **case** statement places the selected statements on a separate line from the **case** labels. The **B** directive (see below) tells the formatter to place these statements on the same line as the **case** labels if the statements will fit.

Similarly, the rules for **if-then-else**, **for**, **while**, and **with** place the controlled statements on separate lines. The **B** directive tells the formatter to place the controlled statement on the same line as the statement header if the statement will fit.

Tables

Many Pascal programs contain lists of initialization statements or constant declarations that are logically a single action or declaration. You may want these to be fit into as few lines as possible. The **S** directive (see below) allows this. If this is used, logical tab stops are set up on the line, and successive statements or constant declarations are aligned to these tab stops instead of beginning on new lines.

At least one blank is always placed between statements or comment declarations, so if tab stops are set up at every character location, statements will be packed on a line.

Structured statements, which normally format on more than one line, are not affected by this directive.

Using PASMAT

Invoke PASMAT with the following command:

```
.R PASMAT  
*output-file = input-file /options="directives"
```

input-file:

The Pascal source file being reformatted. PASMAT accepts only one input file. The default file extension for both input and output is **.PAS**.

output-file:

The reformatted Pascal source file. If *output-file* is omitted, the output file receives the same file name and extension as the input file and becomes the latest version of that file.

options="directives":

Settings for formatting directives. The **options** switch is optional. It may be abbreviated to **o** and may be placed anywhere on the command line. Though the **=** is shown as the switch separator, a colon (**:**) may also be used between the **options** switch and the directives. When specified on the command line, directives must be placed in quotes as shown. The *directives* field will be scanned as though the directives were in a Pascal comment at the start of the source program.

1. The first part of the paper is devoted to a general discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of matter. The problem is to find the conditions under which the structure of matter is stable. This is a problem of great importance in the theory of the structure of matter.

2. The second part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of matter. The problem is to find the conditions under which the structure of matter is stable. This is a problem of great importance in the theory of the structure of matter.

3. The third part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of matter. The problem is to find the conditions under which the structure of matter is stable. This is a problem of great importance in the theory of the structure of matter.

4. The fourth part of the paper is devoted to a detailed discussion of the problem. It is shown that the problem is of great importance in the theory of the structure of matter. The problem is to find the conditions under which the structure of matter is stable. This is a problem of great importance in the theory of the structure of matter.

PASMAT: A Pascal-2 Formatter

Formatting Directives

Formatting directives may be specified either by an **options** switch on the command line or by a special form of the Pascal comment structure.

Formatting directives are of two breeds: switches that turn *on* with the plus sign (+) and *off* with the negative sign (-) (e.g., **R+** and **L-**); or numeric directives of the form **T=5**. Multiple directives are separated by commas (e.g., **R+,L-**). Blanks are not allowed within a directive. Case is ignored: **R+** is the same as **r+** in a directive.

By definition (and by default), certain directives override other directives, such as the **L** directive overriding the **U** and **R** directives. Therefore, when turning on a directive, you must turn off any directive that overrides it. For example, suppose you want all Pascal reserved words in upper case. In addition to setting **R+**, which specifies upper case, you must also turn off the **L** directive with **L-**. See the second example under "PASMAT Examples."

The following example shows a program named **PROG.PAS** being formatted with a command-line directive that sets the switch **B** on, **R** off and the numeric directives **O** to 72 and **T** to 5.

```
.R PASMAT  
*PROG/OPTIONS="B+,O=72,T=5,R-"
```

If used in the program text as part of an embedded Pascal comment, format directives are placed within square brackets that, along with any other comments, are placed within the standard Pascal comment braces. A compiler directive (e.g., **\$nomain**), if present, must begin any comment containing a PASMAT directive. In this case, the PASMAT directive may come before or after any other text:

```
{$compiler-directives text [directives] text}
```

If no compiler directive is present, the PASMAT directive must begin the comment:

```
{[directives] text}
```

The following embedded directive has the same effect as the command-line directive shown above.

```
{[b+,o=72,t=5,r-]}
```

The PASMAT formatting directives are:

- A** (Default **A-**) Adjusts each identifier so that the first instance of the identifier determines the appearance of all subsequent instances of the identifier. This facility standardizes the use of upper-case and lower-case characters and the break character (**_**) in program text. This directive overrides the **U** directive.
- B** (Default **B-**) Specifies that the statements following a **then**, or **else**, **for**, **with** or **while** will be put on the same line if they will fit. The statement following a **case** label will be put on the same line if it fits. The result is a shorter output, which may be easier to read but which also may be harder to correct.
- C** (Default **C-**) Converts leading blanks to tabs on output.
- F** (Default **F+**) Turns formatting on and off. This directive goes into effect immediately after the comment in which it is placed and can save carefully hand-formatted portions of a program.
- K** (Default **K-**) Converts the Pascal-1 **else** clause in a **case** statement to **otherwise** as used in Pascal-2.
- L** (Default **L+**) Specifies that the case of identifiers and reserved words be a literal copy of the input. This directive overrides the **U** and **R** directives and is disabled by the **P+** directive.

- M** (Default **M+**) Converts all alternate symbol representations to the standard form. Otherwise, all symbols are left as they are in the text. The nonstandard comment brackets `/* ... */` are always converted, either to braces or, in the case of **M-**, to `(* ... *)`.
- N** (Default **N-**) Inserts no new lines into the output unless they are required to make the lines fit. This directive just indents the source, keeping the line structure set up by the user. If a line exceeds the output length, it will be broken at the best place available, but the results may not be what you want. Look things over carefully after using this option.
- O** (Numeric directive, default **O=78**) Specifies the width of the output line. The maximum value allowed is 132 characters. If a particular token will not fit in the width specified, the line will be lengthened accordingly, and a message at the end of the formatting will give the number of times the width was exceeded and the output line number of the first occurrence.
- P** (Default **P-**) Sets "portability mode" formatting, which removes break characters (`_`) from identifiers. The first letter of each identifier, and the first letter following each break character, will be made upper case, while the remaining characters will be in lower case. This directive overrides the **L** and **U** directives. The **R** directive sets the case of reserved words.

Warning: Pascal-2 considers break characters significant: `User_DoesThis` is one identifier and `UserDoes_This` is another. Take care when using this directive that you do not make two different identifiers the same: `UserDoesThis` and `UserDoesThis`.
- R** (Default **R-**) Specifies that all reserved words will be in upper case. With this off, reserved words will be in lower case. The **L** directive overrides the **R** directive. When using **R+** you must also use **L-** to turn off the overriding directive. See the second example under "PASMAT Examples."
- S** (Numeric directive, default **S=1**) Specifies the number of statements per line. The space from the current indentation level to the end of the line is divided into even pieces, and successive statements are put on the boundaries of successive pieces. A statement may take more than one piece, in which case the next statement again goes on the boundary of the next piece. This is similar to the tabbing of a typewriter.

Any statement requiring more than one line will not be affected, but may cause unexpected results on following statements. This directive only affects the constant declaration and statement portions of the program and is intended for use in initializing tables. The default value of 1 provides normal formatting.
- T** (Numeric directive, default **T=2**) Specifies the amount to "tab" for each indentation level. Statements that continue on successive lines will be additionally indented by half the value of **T**.
- U** (Default **U-**) **U+** specifies that identifiers are converted to upper case; **U-** specifies that they will be converted to lower case. The **L**, **P** and **A** directives override this directive. When using **U+** you must also use **L-** to turn off the **L** directive. Also, make sure the **P** directive is off (**P-**, the default).

the first of these is the fact that the
the second is the fact that the
the third is the fact that the
the fourth is the fact that the
the fifth is the fact that the
the sixth is the fact that the
the seventh is the fact that the
the eighth is the fact that the
the ninth is the fact that the
the tenth is the fact that the

the eleventh is the fact that the
the twelfth is the fact that the
the thirteenth is the fact that the
the fourteenth is the fact that the
the fifteenth is the fact that the
the sixteenth is the fact that the
the seventeenth is the fact that the
the eighteenth is the fact that the
the nineteenth is the fact that the
the twentieth is the fact that the

the twenty-first is the fact that the
the twenty-second is the fact that the
the twenty-third is the fact that the
the twenty-fourth is the fact that the
the twenty-fifth is the fact that the
the twenty-sixth is the fact that the
the twenty-seventh is the fact that the
the twenty-eighth is the fact that the
the twenty-ninth is the fact that the
the thirtieth is the fact that the

the thirty-first is the fact that the
the thirty-second is the fact that the
the thirty-third is the fact that the
the thirty-fourth is the fact that the
the thirty-fifth is the fact that the
the thirty-sixth is the fact that the
the thirty-seventh is the fact that the
the thirty-eighth is the fact that the
the thirty-ninth is the fact that the
the fortieth is the fact that the

the forty-first is the fact that the
the forty-second is the fact that the
the forty-third is the fact that the
the forty-fourth is the fact that the
the forty-fifth is the fact that the
the forty-sixth is the fact that the
the forty-seventh is the fact that the
the forty-eighth is the fact that the
the forty-ninth is the fact that the
the fiftieth is the fact that the

the fifty-first is the fact that the
the fifty-second is the fact that the
the fifty-third is the fact that the
the fifty-fourth is the fact that the
the fifty-fifth is the fact that the
the fifty-sixth is the fact that the
the fifty-seventh is the fact that the
the fifty-eighth is the fact that the
the fifty-ninth is the fact that the
the sixtieth is the fact that the

the sixty-first is the fact that the
the sixty-second is the fact that the
the sixty-third is the fact that the
the sixty-fourth is the fact that the
the sixty-fifth is the fact that the
the sixty-sixth is the fact that the
the sixty-seventh is the fact that the
the sixty-eighth is the fact that the
the sixty-ninth is the fact that the
the seventieth is the fact that the

the seventy-first is the fact that the
the seventy-second is the fact that the
the seventy-third is the fact that the
the seventy-fourth is the fact that the
the seventy-fifth is the fact that the
the seventy-sixth is the fact that the
the seventy-seventh is the fact that the
the seventy-eighth is the fact that the
the seventy-ninth is the fact that the
the eightieth is the fact that the

the eighty-first is the fact that the
the eighty-second is the fact that the
the eighty-third is the fact that the
the eighty-fourth is the fact that the
the eighty-fifth is the fact that the
the eighty-sixth is the fact that the
the eighty-seventh is the fact that the
the eighty-eighth is the fact that the
the eighty-ninth is the fact that the
the ninetieth is the fact that the

the ninety-first is the fact that the
the ninety-second is the fact that the
the ninety-third is the fact that the
the ninety-fourth is the fact that the
the ninety-fifth is the fact that the
the ninety-sixth is the fact that the
the ninety-seventh is the fact that the
the ninety-eighth is the fact that the
the ninety-ninth is the fact that the
the hundredth is the fact that the

PASMAT: A Pascal-2 Formatter

Limitations and Errors

PASMAT is limited in these ways:

- The maximum input line length is 132 characters.
- The maximum output length is 132 characters.
- Only syntactically correct statements are formatted. A syntax error in the code will cause the formatting to abort. An error message will give the input line number on which the error is detected. The error checking is not perfect, and successful formatting is no guarantee that the program will compile.
- The number of indention levels handled by PASMAT is limited; PASMAT will abort if this number is exceeded — a rare circumstance.
- If a comment will require more than the maximum output length (132) to meet the rules given, processing will be aborted. This situation should be even rarer than indention-level problems.
- When it aborts, PASMAT attempts to copy the rest of the file. You should, however, recover a copy of the source file and inspect the PASMAT-generated copy carefully; we cannot guarantee that PASMAT will recover all the text for every error condition.

PASMAT Examples

To show how the various PASMAT options work together, we will take the sample program that follows and show how it appears after reformatting with two different sets of options.

```

program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!) }
var E, series_term: real; N: integer;
begin
{ set initial conditions }
E := 1.0; N := 1; SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
repeat
E := E + seriesterms;
{ compute next term of series }
N := N + 1; seriesterms := seriesterms / N;
until E = (E + SeriesTerm);
writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
end.

```

First we reformat the program using the standard indentation of text and comments. We use the output directive on the command line to specify the width of the output line, and we specify a short line width to illustrate the right-justification of long comments.

The program is formatted with the commands:

```

.R PASMAT
*EFACT/OPTIONS="0=66"

```

Program text after formatting:

```

program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!) }

var
    E, series_term: real;
    N: integer;

begin
    { set initial conditions }
    E := 1.0;
    N := 1;
    SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
    repeat
        E := E + seriesterms;
        { compute next term of series }
        N := N + 1;
        seriesterms := seriesterms / N;
    until E = (E + SeriesTerm);
    writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
end.

```

The second example illustrates embedded PASMAT commands. We have altered the original program by inserting the text {[A+,L-,R+]} before the first line. The directive A+ changes each identifier to

PASMAT: A Pascal-2 Formatter

match the appearance of the first use of that identifier. (Notice the variant forms of `series_term` and `E` in the original program.) The directives `L-` and `R+` together turn off the literal reproduction of the reserved words and make them upper case. The program is formatted with the commands:

```
.R PASMAT  
*EFACT
```

Program text after formatting:

```
{[A+,L-,R+]}  
PROGRAM Efact(output);  
{ Compute an approximation for E from its Taylor series }  
{ The Nth term in the series is 1/(N!) }  
  
VAR  
    E, series_term: real;  
    N: integer;  
  
BEGIN  
    { set initial conditions }  
    E := 1.0;  
    N := 1;  
    series_term := 1.0;  
    { loop to approximate E; quit when the series sum stops changing }  
    REPEAT  
        E := E + series_term;  
        { compute next term of series }  
        N := N + 1;  
        series_term := series_term / N;  
    UNTIL E = (E + series_term);  
    writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);  
END.
```


PB: A Pascal-2 Formatter

PB is designed on the premise that a formatting program can't do everything, that formatting requires an understanding of the meaning of a program. Thus, PB is meant to assist, rather than replace, the manual arrangement of program format. The simple transformations performed by PB reduce the tediousness of formatting program text and help ensure consistency within a variety of personal formatting styles.

PB can be used on code as it is being developed, even code that is incomplete or incorrect. You write a program, or program fragment, to some level of detail, then run it through PB. You can then edit the resulting code to alter its meaning or improve its appearance, and use PB again. This cycle can be repeated as the code progresses from initial idea to working program, and later as the program is "maintained."

Text produced by PB usually looks much like the input. Each input line is transformed into a single output line containing essentially the same text; within the code on a line the spacing is the same; and simple statements that continue onto multiple lines stay lined up.

PB adjusts program format to be consistent with the syntactic structure of Pascal. Statements at the same nesting level line up, and indention increases with the nesting level. Where possible, trailing comments are lined up with one another. Keywords and identifiers in the text are altered to match the capitalization style of their first occurrence (which may be in an included file).

Using PB

You invoke PB with the following command:

```
.R PB  
*output-file = input-files /switches
```

input-files:

The Pascal source files being reformatted. The default file extension is .PAS. Multiple files, if specified, are separated by commas. Multiple files are concatenated to produce the output file.

output-file:

The reformatted Pascal source file. The default file extension is .PAS. If *output-file* is not specified, the output file will be created with the same file name and extension as the last input file and becomes the latest version of the file.

switches: Command-line switches used to adjust PB formatting. These switches must be placed after the input file names on the command line. *Switches* may be either or both of these:

The *indent:num* switch specifies the number of columns (*num*) in an indention step (the space that text is shifted when the nesting level changes). The default setting is 2; larger values make the separation between levels clearer, but may force text past the right margin.

The *comment:num* switch specifies the column (*num*) to which trailing comments are indented. The default setting is 33; this value works well when trailing comments are used primarily to annotate declarations.

The switches may be abbreviated to two letters. Multiple switches are separated by a slash '/'.

January 22, 1973

Dear Mr. [Name],
I am writing to you regarding the [Topic].
I have received your letter of [Date] and am pleased to hear from you.
The information you provided is very helpful and I am currently reviewing it.
I will get back to you as soon as possible with a response.
Thank you for your patience and understanding.
Sincerely,
[Signature]

I am writing to you regarding the [Topic].
I have received your letter of [Date] and am pleased to hear from you.
The information you provided is very helpful and I am currently reviewing it.
I will get back to you as soon as possible with a response.
Thank you for your patience and understanding.
Sincerely,
[Signature]

PB: A Pascal-2 Formatter

Example

This example illustrates the functions provided by PB. The example also shows a particular development style, discussed above, to which PB is suited.

We start with the program at an intermediate stage in its development. Some new code has just been added. Notice that the new code is not indented; it is broken into reasonable lines, but we will let PB do the rest of the formatting.

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
    repeat
      x := x + 1;
      prim := x is a prime number;
    until prim;
    write(X);
  end;
end.
```

Processing by PB gives this result:

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
    repeat
      X := X + 1;
      prim := X is a prime number;
    until prim;
    write(X);
  end;
end.
```

The indent changes at most one step from line to line. When control constructs appear one per line (the **repeat** statement, for instance) each causes the indent to increase by one step, but when a second one appears on a line it has no effect on indentation. In the example, this has been used to cut the "noise" from **begin...end** brackets.

Notice a couple of things at this stage. First, PB does not know much about Pascal language syntax (note the phrase "**X is a prime number**"). Second, PB uses the first instance of a word, regardless of context, for its capitalization style (you can set the style for an identifier by adjusting its declaration, since the declaration of an identifier must precede its use).

Example

Of course, the program is not yet complete, nor does it contain any comments. The following is closer to a final version.

```
type Index = 1..n;
var
  X: integer; { number being tested for primality }
  j, { count of primes found }
  k, { trial divisor }
  lim: index; { last divisor to test }
  Prim: boolean; { 'true' until a divisor is found }
  P: array[Index] of integer; { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; Lim := 1;
  write('2');
  for J := 2 to n do begin
    repeat
      X := X + 2;
    if sqrt(P[Lim]) <= X then Lim := Lim + 1;
    K := 2; Prim := true;
    while Prim and (K < Lim) do begin
      Prim := (X mod P[K]) <> 0;
      K := K + 1;
    end;
    until Prim;
    P[J] := X; write(X);
  end;
end.
```

Processing by PB gives:

```
type Index = 1..n;
var
  X: integer;           { number being tested for primality }
  j,                    { count of primes found }
  k,                    { trial divisor }
  lim: Index;           { last divisor to test }
  Prim: boolean;        { 'true' until a divisor is found }
  P: array[Index] of integer; { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; lim := 1;
  write('2');
  for j := 2 to n do begin
    repeat
      X := X + 2;
      if sqrt(P[lim]) <= X then lim := lim + 1;
      k := 2; Prim := true;
      while Prim and (k < lim) do begin
        Prim := (X mod P[k]) <> 0;
        k := k + 1;
      end;
    until Prim;
    P[j] := X; write(X);
  end;
end.
```


The comments have been moved to the right, where they stand apart from the program code and line up for easier reading. If we had wanted to keep a comment attached to the code, we could have placed it in front of the final comma or semicolon on its line (then it would not be a trailing comment and would be treated as part of the text), or placed it on a line by itself (where it would be aligned at the prevailing indent).

Detailed Formatting Rules

Indentation is directed by the nesting of control constructs in the program text. Generally, when the nesting level increases, the indent increases by the indentation step; when a nesting level ends, the indent reverts to that of the surrounding nesting level. A change of indentation at the beginning of a line takes effect immediately; otherwise, it takes effect on the next line. The exceptions to the rules are:

- The start of a new nesting level does not change the indent if it begins on the same line as the surrounding level.
- When a line begins with a statement label, the indent for that line is decreased by the indentation step.

Normal indentation rules do not apply when a simple statement or clause continues across multiple lines. In these cases, the initial line is indented normally, but the following lines are arranged to preserve their alignment with the initial line. Changes of indentation within continued lines take effect after the last continuation.

A similar adjustment occurs when a comment continues across multiple lines. When a trailing comment is continued, the following lines stay aligned with the initial part of the comment, but not necessarily with the rest of that line (a trailing comment may shift in relation to the rest of the line).

The following constructs affect the indentation level:

- A *program-declaration*, *procedure-declaration* or *function-declaration* is arranged so that the *heading*, the keyword introducing a *label-declaration-part*, *const-definition-part*, *type-definition-part*, or *variable-declaration-part*, and the *body* are all at the same indentation level. The list of declarations within a *label-declaration-part*, *const-definition-part*, *type-definition-part*, *variable-declaration-part*, or *procedure-and-function-declaration-part* is set one indentation step deeper.
- The *component-type* of an *array-type* or *file-type*, and the *base-type* of a *set-type* are indented one more step. Within a *record-type* the *field-list* is indented another step, the list of *variants* within the *variant-part* is indented an additional step, and the *field-lists* within individual *variants* are indented yet another step.
- The *statement-sequence* within a *compound-statement* is indented an additional step.
- The controlled *statement* within a *for-statement*, *if-statement*, *else-part*, *while-statement* or *with-statement*, and the *statement-sequence* within a *repeat-statement* are indented another step. Within a *case-statement* the list of *case-list-elements* is indented one more step, and the controlled *statement* of each *case-list-element* is indented an additional step.

The first part of the document is a letter from the President of the United States to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The second part of the document is a letter from the President to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The third part of the document is a letter from the President to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The fourth part of the document is a letter from the President to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The fifth part of the document is a letter from the President to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

The sixth part of the document is a letter from the President to the Congress, dated January 1, 1861. It is a very important document, as it is the first official communication of the new President to the new Congress.

XREF: A Pascal-2 Cross-Reference Lister

XREF produces a cross-reference listing of the identifiers in a Pascal program. XREF is helpful when debugging new programs or when modifying existing ones. The output shows the use of each identifier in the program, which is beneficial when you're working with medium to large programs.

Each identifier is listed, along with an entry for each reference to that identifier. Each entry consists of the line on which the reference occurs, plus an indication of whether the reference is a declaration or assignment.

Using XREF

You invoke XREF with the following command:

```
.R XREF  
* output-file = input-file /switches
```

input-file:

The Pascal source file being cross-referenced. The input file has a default extension of .PAS. XREF accepts only one input file.

output-file:

The cross-reference file. The output file has a default extension of .CRF. *Output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the input file, and having the default extension, is placed in the default directory.

switches: Command-line switches. *Switches* may be either or both of these:

The **list** switch generates a listing of the input file before the cross-reference. This listing includes line numbers and a flag character (c) indicating multiple line comments and strings. The flag character makes it easier to locate certain bugs that cannot be easily diagnosed by the compiler.

The **width:num** switch specifies the page width for the cross-reference listing, where *num* is the number of characters across. The default is 132.

The switches may be abbreviated to one letter. Multiple switches are separated by a slash '/'.

Limitations

The XREF program has two limitations on the size of the programs it can handle.

- An internal limit exists for the number of distinct identifiers allowed. You can change this number in the XREF source file and recompile the program.
- The total number of references is limited by the amount of dynamic storage available.

The XREF program does not perform a complete syntax analysis of the program, and it may not flag all declarations or assignments.

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

NEW YORK: PUBLISHED BY J. B. LIPPINCOTT & CO.

1884

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

NEW YORK: PUBLISHED BY J. B. LIPPINCOTT & CO.

1884

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

XREF: A Pascal-2 Cross-Reference Lister

Example of XREF Listing

This example shows the cross-referencing of the program EFACT to produce the output file EFACT.CRF.

.R XREF

*EFACT/LIST/WIDTH:66

```
1 program Efact(output);
2 { Compute an approximation for E from its Taylor series.
c 3   The Nth term in the series is 1/(N!).
c 4 }
5
6   var
7     E, SeriesTerm: real;
8     N: integer;
9
10  begin
11    { set initial conditions }
12    E := 1.0;
13    N := 1;
14    SeriesTerm := 1.0;
15    repeat { loop to approximate E; quit when sum stops changing }
16      E := E + SeriesTerm;
17      N := N + 1;
18      SeriesTerm := SeriesTerm / N;
19    until E = (E + SeriesTerm);
20    writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
21  end.
```

Cross reference: * indicates definition, = indicates assignment

-E-

E	7*	12=	16=	16	19	19	20
EFACT	1*						

-I-

INTEGER	8						
---------	---	--	--	--	--	--	--

-N-

N	8*	13=	17=	17	18	20	
---	----	-----	-----	----	----	----	--

-O-

OUTPUT	1*						
--------	----	--	--	--	--	--	--

-R-

REAL	7						
------	---	--	--	--	--	--	--

-S-

SERIESTERM	7*	14=	16	18=	18	19	
------------	----	-----	----	-----	----	----	--

-W-

WRITELN	20						
---------	----	--	--	--	--	--	--

end xref 8 identifiers 24 total references

PROCREF: Pascal-2 Procedural Cross-Reference Lister

PROCREF, based on a procedural cross-reference program published by Arthur Sale in *Pascal News* (Number 17, March 1980), is designed to help programmers sort through the procedures in medium to large Pascal programs. The program has been modified to allow the use of multiple input files and `%include` directives and to provide "called by" data in the listing.

PROCREF provides a quick overview of the procedural organization of a program, which is beneficial when you're working with medium to large programs. The PROCREF utility reads the text of a Pascal program to produce a compact listing of the procedure headings and an alphabetized list of procedures with usage information. PROCREF processes `%include` directives in the same way as the Pascal-2 compiler, so that all parts of a compilation can be analyzed.

The procedure listing includes each procedure heading, along with its location in the input file. Procedure headings are indented to show lexical level. No attempt is made to fit the procedure headings into a limited line width.

The cross-reference listing places procedures in alphabetical order. For each procedure the listing includes:

- The file and line number where its heading starts.
- The file and line number where its body starts, unless it is external or is a formal procedure parameter and has no body. In such a case, the note **external** or **formal** is printed.
- If the procedure was declared **forward** or is externally defined, the listing contains the file and line number where the procedure heading stub starts.
- A list of all procedures immediately called by this procedure. These are listed in the order in which they occur in the text. A procedure is listed only once, even if it is called more than once.
- A list of all procedures that call this procedure. Again, the list is in textual order and only one reference is shown per procedure.

Only the first sixteen characters of a procedure name appear in the cross-reference listing. Those characters are written exactly as they appear in the program text.

Using PROCREF

You invoke PROCREF with the following command:

```
.R PROCREF  
* output-file = input-files /width:num
```

input-files:

The Pascal source files being cross-referenced. The input files have a default extension of `.PAS`. Multiple input files, if specified, are separated by commas. Multiple files will be concatenated.

output-file:

The cross-reference file. The output file has a default extension of `.PRF`. The *output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the last input file, and having the default extension, is placed in the default directory.

width:num

Specifies the page width for the cross-reference listing, where *num* is the number of characters across the page. The default is 80 characters. The **width** switch is optional and may be abbreviated to one letter.

1. The first part of the paper is devoted to a general discussion of the problem.

2. The second part is devoted to a detailed analysis of the various factors involved.

3. The third part is devoted to a discussion of the results of the analysis.

4. The fourth part is devoted to a discussion of the conclusions of the study.

5. The fifth part is devoted to a discussion of the implications of the study.

6. The sixth part is devoted to a discussion of the limitations of the study.

7. The seventh part is devoted to a discussion of the future research.

8. The eighth part is devoted to a discussion of the significance of the study.

9. The ninth part is devoted to a discussion of the conclusions of the study.

10. The tenth part is devoted to a discussion of the implications of the study.

PROCREF: Pascal-2 Procedural Cross-Reference Lister

Note that the RT-11 system will automatically truncate the name PROCREF to PROCRE.

Limitations

The PROCREF program does not do a complete syntax analysis of the program being processed. PROCREF will err in one case: If a field identifier in a record has the same name as a procedure, and if that field is referenced without a preceding record variable name, as in a with statement, the field identifier will be treated as a reference to the procedure.

Example

Let's assume that we wish to generate a procedure cross-reference for the following program, LVSPPOOL.PAS.

Pascal-2 RT11 SJ V2.1A 5-Aug-83 7:04 PM Site #1-1 Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 228-7760
LVSPPOOL/LIST

```
1      program LVSpool(input, output);
2          procedure ScanLV; external;
3          procedure ReadFontInfo(i: integer; j: integer); forward;
4
5          procedure LoadFonts;
6              procedure GetByte;
7                  begin
8                      end;
9              begin
10                 GetByte;
11                 ReadFontInfo(1, 2);
12             end;
13
14         procedure ReadFontInfo;
15             begin
16                 LoadFonts;
17             end;
18
19         procedure ShowPage;
20             begin
21                 ScanLV;
22             end;
23
24     begin                main program
25         ReadFontInfo(0,1);
26         ShowPage;
27     end.
```

*** No lines with errors detected ***

We cross-reference the procedures as follows.

.R PROCREF
*LVSPPOOL=LVSPPOOL/W:72

The W:72 requests that the cross-reference listing not exceed 72 characters in width so that the result may be printed on a terminal. The result, placed in the file LVSPPOOL.PRF, consists of:

Procedural Cross-Referencer - Version 3.0
 LVSPPOOL/W:72

Line Program/procedure/function heading

LVSPPOOL.PAS:

```

1  program LVSpool(input, output);
2      procedure ScanLV; external;
3      procedure ReadFontInfo(i: integer; j: integer); forward;
5      procedure LoadFonts;
6          procedure GetByte;
14     procedure ReadFontInfo;
19     procedure ShowPage;
```

Procedural Cross-Referencer - Version 3.0
 LVSPPOOL/W:72

Cross Reference Listing

GetByte	Head: LVSPPOOL.PAS, 6	Body: LVSPPOOL.PAS, 7
Called by	LoadFonts	
LoadFonts	Head: LVSPPOOL.PAS, 5	Body: LVSPPOOL.PAS, 9
Calls	GetByte	ReadFontInfo
Called by	ReadFontInfo	
LVSpool	Head: LVSPPOOL.PAS, 1	Body: LVSPPOOL.PAS, 24
Calls	ReadFontInfo	ShowPage
ReadFontInfo	Head: LVSPPOOL.PAS, 3	Body: LVSPPOOL.PAS, 15
	Forward, header stub: LVSPPOOL.PAS, 14	
Calls	LoadFonts	
Called by	LoadFonts	LVSpool
ScanLV	Head: LVSPPOOL.PAS, 2	external
Called by	ShowPage	
ShowPage	Head: LVSPPOOL.PAS, 19	Body: LVSPPOOL.PAS, 20
Calls	ScanLV	
Called by	LVSpool	

Dynamic String Package

The Pascal standard implements character strings in two ways: as a sequence of two or more characters between single-quote marks (a literal string); or as a packed array of `char` (a variable string). However, the standard does not provide adequate facilities for manipulating character strings and only allows assignments of one string to another string and comparisons of two strings of equal length.

Pascal-2's Dynamic String Package extends the meager string-handling capabilities of standard Pascal, providing the ability to perform sophisticated operations on strings of varying lengths. The string package, `STRING`, is a collection of string-processing procedures and functions that allows Pascal programs to read and write strings, concatenate two strings, search one string for another, insert one string into another and delete one string from another, assign the value of one string to another string and other string operations. The string package is written in standard Pascal to take advantage of conformant array parameters, which facilitate the passing of variable-length arrays (strings), and to provide portability to other Pascal implementations.

To use the string package, declare string variables as packed arrays of characters with a lower bound of 0 and an upper bound equal to the maximum length for that particular string, as shown:

```
var
  string-name: packed array [0..max-len] of char;
```

where *string-name* is the identifier associated with the string variable and *max-len* is the maximum length of the string in bytes. The actual length of the string is stored in element 0. The characters making up the string are stored starting at element 1. *Max-len* must be greater than 0 and no larger than 255.

The maximum length of a string may be different for each string, depending on the intended use of the string. The string package's use of conformant array parameters makes this possible. Examples:

```
var
  NameString: packed array [0..25] of char;
  SiteNo: packed array [0..7] of char;
  LineOfInput: packed array [0..80] of char;
```

As an alternative, these routines also accept parameters of type `packed array [1..max-len] of char`, where *max-len* is the actual length of the string. Literal strings are of this type. This means you may pass a literal string to any of these procedures as long as the formal parameter is not a `var` parameter.

`STRING` may be included in program source files in one of two ways: in the program code, place the `%include` compiler directive; or on the command line, concatenate `STRING.PAS` with the rest of the source files making up the program ("source concatenation"). We recommend the use of the `%include` directive (e.g., `%include 'string';`). However, if you concatenate the string package with the source file, the command to compile program `PROG` is:

```
.R PASCAL
*STRING.PROG
```

Source concatenation may be used only if the main program does not contain a `program` statement; otherwise, compilation errors result. Refer to "Multiple Source Files" in the Programmer Reference for more information on the `%include` directive.

THE HISTORY OF THE

First part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The second part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The third part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The fourth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The fifth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The sixth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The seventh part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The eighth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The ninth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The tenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The eleventh part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The twelfth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The thirteenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The fourteenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The fifteenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The sixteenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The seventeenth part of the history of the world, from the beginning of time to the present day, as recorded in the Bible and other ancient and modern authors.

The Procedures and Functions

In the definitions below, *string* and *target* represent string variables similar to the previous examples. *File* must be a variable of type *text*. *Start* and *span*, of type *integer*, represent character positions and character ranges, respectively. *Max-len* is the upper boundary, or maximum length, of the array. *Char* may be a variable of type *char* or a literal string of one character.

The string package contains these procedures and functions:

Len(*string*)

An integer function, returns the actual length of *string*. *String* may be a literal string.

Clear(*string*)

Initializes *string* to empty.

ReadString(*file*, *string*)

Reads *string* from *file*. The string is terminated when *eola(file)* becomes true, and a *readln(file)* is performed. Overflow results in truncation to *max-len* characters.

WriteString(*file*, *string*)

Writes *string* to *file*. This procedure does not accept literal strings as parameters. Use *writeln* to terminate a written string manually.

Concatenate(*target*, *string*)

Appends *string* to *target*. The resulting value is *target*. *String* may be a literal string. Overflow results in truncation to *max-len* characters.

Search(*string*, *target*, *start*)

Searches *string* for the first occurrence of *target* to the right of position *start* (characters are numbered beginning with 1). The *Search* function returns the position of the first character in the matching substring, or the value zero if *target* does not appear in *string*. *String* and *target* may be literal strings.

Insert(*target*, *string*, *start*)

Inserts *string* into *target* at position *start*. Characters are shifted to the right as necessary. Overflow produces a truncated *target* of *max-len* characters. The insertion is skipped if the *start* position causes a non-contiguous string. *String* may be a literal string.

Assign(*target*, *string*)

Assigns *string* to *target*. This procedure is especially useful for assigning a literal *string* to a variable string (*target*). To assign one character to a variable string, use the *Asschar* procedure, below.

Asschar(*target*, *char*)

Assigns *char* to *target*. *Char* may be a literal character or a variable name. This procedure is more efficient than procedure *Assign* for the creation of one-character strings. (With *Assign*, a one-character string must first be created as input to *Assign*, which then assigns the character to a variable string.)

Equal(*target*, *string*)

Determines whether *target* is element-for-element identical to *string*. This boolean function returns a *true* value if the two strings are equal, *false* if the two strings are different. *Target* and *string* may be literal strings.

The *start* and *span* parameters in the *Delstring* and *Substring* procedures define a substring beginning at position *start* (between characters *start-1* and *start*) with a length of *abs(span)*. If


```

procedure GetDirective(I: integer);
var
  Ch: char;

begin { GetDirective }
  Clear(Directive); ----- defined in STRING
  while I <= Len(Line) do begin { not end of line yet }
    Ch := Line[I]; I := I + 1;
    if Ch in Letters then begin { get next char of directive }
      Directive[0] := succ(Directive[0]);
      Directive[Len(Directive)] := Ch
    end
    else I := Len(Line) + 1;
  end;
  DirectiveFound := Len(Directive) > 0
end; { end GetDirective }

begin { DirectiveList }
  write('PROSE FILE: ');
  readln(Name);
  reset(Prosefile, Name, '.prs');
  rewrite(Directivefile, '.dtv', Name);
  Assignchar(Escape, '.'); ----- defined in STRING
  Linenum := 0;
  Letters := ['A'..'Z', 'a'..'z']; { Any others end directive }
  while not eof(Prosefile) do begin
    ReadString(Prosefile, Line); ----- defined in STRING
    Linenum := Linenum + 1;
    if Line[1] = Escape[1] then begin { first character is an escape }
      Index := 0; NoneYet := true;
      repeat { find all occurrences of escape characters }
        Index := search(Line, Escape, Index + 1); -- defined in STRING
      until Index <> 0 then begin
        GetDirective(Index + 1); { get the next directive }
        if DirectiveFound then begin
          Assign(Outline, Escape); ----- defined in STRING
          Concatenate(Outline, Directive); ----- defined in STRING
          if NoneYet then begin
            write(Directivefile, ' Line ', Linenum: 4, ' ');
            NoneYet := false
          end
          else write(Directivefile, ' ');
          WriteString(Directivefile, Outline); -- defined in STRING
          writeln(Directivefile);
        end;
      end;
    until Index = 0;
  end;
end. { DirectiveList }

```

For this illustration, PDLIST reads the PROSE input file presented in Appendix A of "PROSE: A Text Formatter," Example 2, later in this guide. The name of the input file is PEXAM2.PRS. To

Dynamic String Package

see what the input looks like, refer to the aforementioned example in the PROSE section.

Run PDLIST using these commands:

.R PASCAL

*PDLIST

.LINK PDLIST,SY:PASCAL

.RUN PDLIST

PROSE FILE: PEXAM2

Output — the list of directives used in the file — is written to PEXAM2.DTV, which looks like this:

Line	1	.COMMENT
Line	2	.INPUT
Line	3	.OPTION
Line	4	.FORM
Line	5	.MARGIN
Line	6	.PARAGRAPH
Line	21	.OPT
Line	22	.MAR
Line	23	.PARAGRAPH
Line	28	.OPT
		.MAR
		.PAR

MACRO-11 Procedures With Pascal-2

Although most programs can be written within the Pascal-2 language, applications involving interface to the operating system require the use of MACRO-11 assembly language code. A set of macros provided with the Pascal-2 system makes this interface easy. You can code a set of macro calls that look much like a Pascal procedure declaration, and the PASMAL macro package will assign addresses to the parameters and generate procedure entry and exit code.

Design of MACRO-11 Procedures

Follow these general rules in deciding what to put in a MACRO-11 procedure:

- Do the absolute minimum in MACRO-11. If you must use MACRO-11 code to use a system service, process the result in Pascal-2 code. (This is not always possible, since some operating systems require very low-level manipulations.)
- Isolate a common function and make the procedure handle the most general case of that function.
- Pass all data to and from the procedure as parameters. Global references from MACRO-11 are not recommended for these reasons: the address is hard to find; if the Pascal program changes, the MACRO program will have to be changed; and global references cannot be checked for type compatibility. This guide does not describe ways to make global references.

Once you have decided on the contents of the procedure, define the calling sequence as a Pascal external procedure. Then write a functional description of the procedure. Then actually write the procedure. These documents will be your implementation guide.

When you have the external definition, use the PASMAL macro package described below to define parameters and local variables. As long as the stack is not changed within the procedure, these macros can access parameters or local variables directly. For this reason, you should probably store local temporary values in the local variables rather than pushing them on the stack. If thoroughly familiar with writing MACRO-11 code, you can use the stack, but make sure you understand the Pascal-2 run-time structure, described in the Programmer's Guide.

MACRO-11 Procedures With Pascal-2

The PASMAL Macro Package

The PASMAL macro package is provided to simplify the writing of MACRO-11 procedures to interface with Pascal-2. Using this package, you can declare procedures, parameters, and variables, and you can easily refer to these items within the procedure.

The package consists of the following macros:

<u>Name</u>	<u>Arguments</u>	<u>Function</u>
proc	<i>procname</i>	Begin the declaration for the procedure <i>procname</i> .
func	<i>funcname</i> <i>result</i> <i>restype</i>	Begin the declaration for the function <i>funcname</i> . The returned value will be that assigned to <i>result</i> , of type <i>restype</i> .
param	<i>parmname</i> <i>parmtyp</i>	Declare a parameter named <i>parmname</i> of type <i>parmtyp</i> .
var	<i>varname</i> <i>vartyp</i>	Declare a local variable named <i>varname</i> of type <i>vartyp</i> .
save	<reg0, ... , regn>	Specify general registers to save on procedure entry.
rsave	<ac0, ... , acn>	Specify floating accumulators to save on procedure entry.
begin		Begin the actual procedure code. This macro generates code to push the variables on the stack and to save registers.
endpr		End the code for this procedure, restore registers, pop variables and parameters from the stack, and return to the calling location.

The following example demonstrates how these macros may be used in a procedure definition. Note the correspondence between the Pascal-2 code and the MACRO-11 code.

Pascal-2 procedure definition:

<code>procedure Examl(Inp1: integer;</code>	<code>{ first value parameter }</code>
<code> Inp2: real;</code>	<code>{ second value parameter }</code>
<code> var Outp: integer</code>	<code>{ variable parameter };</code>
 <code>var</code>	
<code> Var1: integer;</code>	<code>{ first local variable }</code>
<code> Arr1: array [1..3] of integer;</code>	<code>{ second local var }</code>
 <code>begin</code>	<code>{ begin body of procedure }</code>
<code> :</code>	
<code>end;</code>	<code>{ end of procedure }</code>

_____ procedure code

Dear Mr. [Name],

I have received your letter of the 10th inst.

and am sorry to hear that you are

not well.

I hope you will soon be

able to return to your

work.

Yours truly,

[Signature]

The corresponding MACRO-11 code:

```

proc      exampl          ; declare the procedure
param    inp1, integer    ; first value parameter
param    inp2, real       ; second value parameter
param    outp, address    ; variable parameter

var       var1, integer;   ; first local variable
var       arr1, 3*integer  ; second local variable

save      <r0,r1>          ; registers being used
rsave     <ac0,ac1>        ; floating accum being used

begin                                           ; begin body of code
:
----- procedure code
endpr                                           ; reset everything and return

```

Using PASMAL

The macros described in the following sections are included in the file PASMAL.MAC, which also includes definitions of standard data types. It is assumed that this file will be assembled as a header to any MACRO-11 code. This would normally be done with a command line similar to:

```

.R MACRO
*EXTPRO, EXTPRO=SY: PASMAL, EXTPRO

```

The result of this assembly is an object file (.OBJ) that is linked in the same way as any other external module.

MACRO-11 modules assemble with the PASMAL package are referenced from Pascal via the **external** directive instead of the **nonpascal** directive, because PASMAL simulates the Pascal calling sequence. (MACRO-11 routines assembled without the PASMAL package can be referenced via the **nonpascal** directive.) For example:

```

procedure ExtProc(Parm1: integer);
external;

```

For details, see "External Modules" in the Programmer's Guide.

The example command line above also generates a listing file (.LST). Listing of the PASMAL file is disabled with a **.NLIST** directive at the start of the file. A compensating **.LIST** directive is placed at the end of the file, so a program listing is not affected. Defining the tag **\$LIST** anywhere in your code will enable listing of the PASMAL file.

The macros depend on the existence of a uniform radix throughout the declaration of a single procedure. This radix may be octal or decimal, but it must not be changed within a procedure declaration. Also, the macros use labels of the form **Q\$xxx** and macros of the form **\$Pxxx** for storing state data. Avoid such forms in your own code.

Handwritten text, mostly illegible due to fading. Appears to be a letter or document with several paragraphs.

Handwritten text, mostly illegible due to fading. Appears to be a letter or document with several paragraphs.

Handwritten text, mostly illegible due to fading. Appears to be a letter or document with several paragraphs.

MACRO-11 Procedures With Pascal-2

Procedure Definition Macros

The PASMAC procedure definition macros must be used in the order:

<u>Macro</u>	<u>Usage</u>
proc/func	Exactly one of these is required
param	As many as required (or none)
var	As many as required (or none)
save/rsave	Either or both as needed
begin	Required
:	User code
endpr	Required

A MACRO-11 error is detected if the macro calls are not made in the required order.

Above references to parameter and variable "types" assume that "type" identifiers are equivalent to the length of a value of that type. For example, the identifier **integer** has the value of 2, the identifier **real** has the value of 4, and a disk buffer may have the value of 512. The PASMAC package defines some standard types. See "Type Definitions" below.

Parameter, variable and function result names are set to offsets relative to the value of the stack pointer at the end of the **begin** macro. This takes into account local variables allocated on the stack, plus the space used for register saving. You must take into account any additional values you push onto the stack.

Examples:

```
param    param1, integer    ; defines param1
:
mov      param1(sp), r0      ; use param1
```

The 'Proc' Macro

The **proc** macro, used to begin the definition of a procedure, specifies the name to be used and initializes the symbols that store data about the procedure. This macro must be the first macro used in a procedure declaration.

The calling sequence is:

```
proc     procname[, check=1]
```

where

procname

is the name to be used to call the procedure. Only the first six characters of this name are significant.

check

is an optional parameter specifying stack overflow checking. A non-zero value (default) requests a stack overflow check. This check is free (and always done) if more than three registers are saved, and costs two words in the procedure entry otherwise. The time for the check is very small, so disabling it is not recommended.

Examples:

```
proc     p, check=0
or:
proc     p, 0
```


Begins the declaration of a procedure with the external name **p** and no stack overflow checking.

```
proc    savetime
```

Begins the declaration of a procedure with the external name **saveti** and stack checking enabled.

The 'Func' Macro

The **func** macro, similar in function to the **proc** macro, also allows you to specify a name and type for the returned value. In Pascal, the returned value is specified by assignment to the function name. In MACRO-11, this assignment is not possible, since the function name is used for the procedure entry and cannot also point to the appropriate place on the stack. Any value assigned to the result name defined in the **func** macro at exit from the function is returned as the function value.

The calling sequence is:

```
func    funcname, resname, restype[, check=1]
```

where

funcname

is the name to be used to call the function. Only the first six characters are significant.

resname

is the name to be used to reference the returned value. Any value assigned to this location during execution is returned to the calling program upon exit from the procedure.

restype

is the length of the result value. This is not used in the current implementation of the macros, but is included for documentation and possible future use.

check

is an optional parameter that enables stack checking if non-zero. See the description under the **proc** macro.

Example:

```
func    curtime,tval,real
```

Begins the declaration of a function with the external name **curtim** and stack overflow checking enabled. The result location will be named **tval**, of type **real**. Here **real** is assumed to have the value 4, which is the length of a single-precision real value.

The 'Param' Macro

The **param** macro specifies parameters to the current procedure or function. Each parameter has one **param** macro, in the order declared in the Pascal procedure declaration. In the Pascal-2 calling sequence, parameters are pushed onto the stack in the order in which they are declared, so the first parameter is at a higher address than the last parameter. Value parameters have the actual value pushed, and variable parameters have the address of the variable pushed. When these parameters are declared, the parameter name is set equal to the offset of that parameter relative to the stack pointer (**sp**) after the **begin** macro has been called. This value may be used to access the parameter location relative to the stack pointer.

The calling sequence is:

```
param    paramname, paramtype
```

1890

My dear Mr. [Name]

I have just received your letter of the 10th inst.

and am glad to hear from you.

I am well and hope this finds you the same.

I have not much news to write at present.

I am, however, very busy with my work.

I will write again when I have more news.

I am, dear Mr. [Name], very respectfully,

Your obedient servant,

[Signature]

[Address]

[City]

[State]

[Country]

MACRO-11 Procedures With Pascal-2

where

paramname

is the name to be used for accessing the parameter. Within the body of the procedure, if the stack pointer (*sp*) has not changed since the **begin** macro, value parameters can be referred to by *paramname(sp)*, and variable parameters can be referred to as *@paramname(sp)*.

paramtype

is the data type used to determine the space on the stack used by this parameter.

Examples:

```
param  input, integer    ; input: integer
param  result, address   ; var result: integer
```

These macros define two parameters. The first is a value parameter with the name **input** of type **integer** and is referred to in the body of the procedure as *input(sp)*. The second is a variable parameter with the name **result** of type **integer**. Note that the type is defined only in the comment; the actual value pushed on the stack is of type **address**. Within the body of the procedure this is *@result(sp)*.

The 'Var' Macro

The **var** macro, similar to the **param** macro, defines a local variable to be allocated on the stack upon procedure entry. The space for these variables is allocated automatically by the **begin** macro, but is not initialized. Such variables are referenced relative to the stack pointer (*sp*).

The calling sequence is:

```
var      varname, vartype
```

where

varname is the name to be used for accessing the variable. Within the body of the procedure, if the stack pointer (*sp*) has not been modified since the **begin** macro, variables can be referred to by *varname(sp)*.

vartype is the data type used to determine the space to be allocated for this variable.

Example:

```
var      temp, integer    ; temp: integer;
var      name, 10*char     ; name: array [1..10] of char;
```

The example defines two local variables. The space for these variables will be pushed onto the stack by the **begin** macro. The variable **temp** has two bytes allocated and is referred to as *temp(sp)*. The variable **name** has ten bytes allocated and is referred to as *name(sp)*.

The 'Save' Macro

The **save** macro specifies the general registers to be saved on procedure entry. The Pascal-2 calling conventions require a procedure to save and restore all registers used within a procedure, so any registers altered within the procedure should be listed here. If more than three registers are to be saved, a routine from the Pascal support library is used to save the registers. The stack pointer and program counter (*sp* and *pc*) cannot be saved.

The calling sequence is:

```
save    <reg1, ..., regn>
```

My dear Mr. [Name],

I have just received your letter of the 14th inst.

and am glad to hear from you.

I am well and hope this finds you the same.

I have not much news to write at present.

I am, however, very much interested in the progress of the [Project].

I am, Sir, very respectfully,
Your obedient servant,

[Signature]

[Address]

where `<reg1, ..., regn>` is a list of registers to be saved, enclosed in angle brackets (`<>`) and separated by commas. These registers will be saved on entry and restored on exit. The registers `sp` and `pc` cannot be saved, as they are modified by the action of saving them.

Examples:

```
save    <r0,r1>
```

Save registers `R0` and `R1` and restore them on exit. The code generated uses explicit `mov` instructions to do this.

```
save    <r0,r1,r2,r3,r4,r5>
```

Save and restore all available registers. Support routines will be used.

The 'Rsave' Macro

The `rsave` macro is useful only for machines with the Floating Point Processor (FPP) hardware option and serves the same function as `save` except for the floating-point accumulators. You are required to specify the FPP mode, either single or double (default is single). Since the accumulators `AC4` and `AC5` cannot be moved directly to memory, they may not be used unless one of the accumulators `AC0` to `AC3` is also used. Of course, you cannot get data into `AC4` or `AC5` without using one of the lower accumulators, so you should not have any problems meeting this requirement.

The calling sequence is:

```
rsave    <accum1, ..., accumn>[, double=0]
```

where

```
<accum1, ..., accumn>
```

is a list of accumulators to be saved, enclosed in angle brackets (`<>`) and separated by commas. These registers will be saved on procedure entry and restored on procedure exit.

`double` is an optional parameter that specifies the saving of two-word accumulators. If set to 1, specifies that the FPP is in double mode. The default is zero. The setting does not affect the setting of the FPP; it simply allows the correct computation of the space required for the registers.

Examples:

```
rsave    <ac0,ac4>
```

Save accumulators `AC0` and `AC4` and assume that the FPP is in single mode.

```
rsave    <ac0>,double=1
```

or:

```
rsave    ac0,1
```

Save accumulator `AC0` and assume that the FPP is in double mode.

The 'Begin' Macro

The `begin` macro marks the start of the procedure body. This and the `endpr` macro are the only ones to actually generate code. When the `begin` macro is assembled, all of the data saved up by the previous macros is used to generate procedure entry code and define all of the parameter and variable addresses.

The calling sequence is:

```
begin
```

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the matter of the

above mentioned case, and in reply to inform you that the same has been forwarded to the proper authorities for their consideration.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

Enclosed for you are the papers in this case, which I have been instructed to forward to you.

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

I am, Sir, very respectfully,
Your obedient servant,

J. H. [Signature]

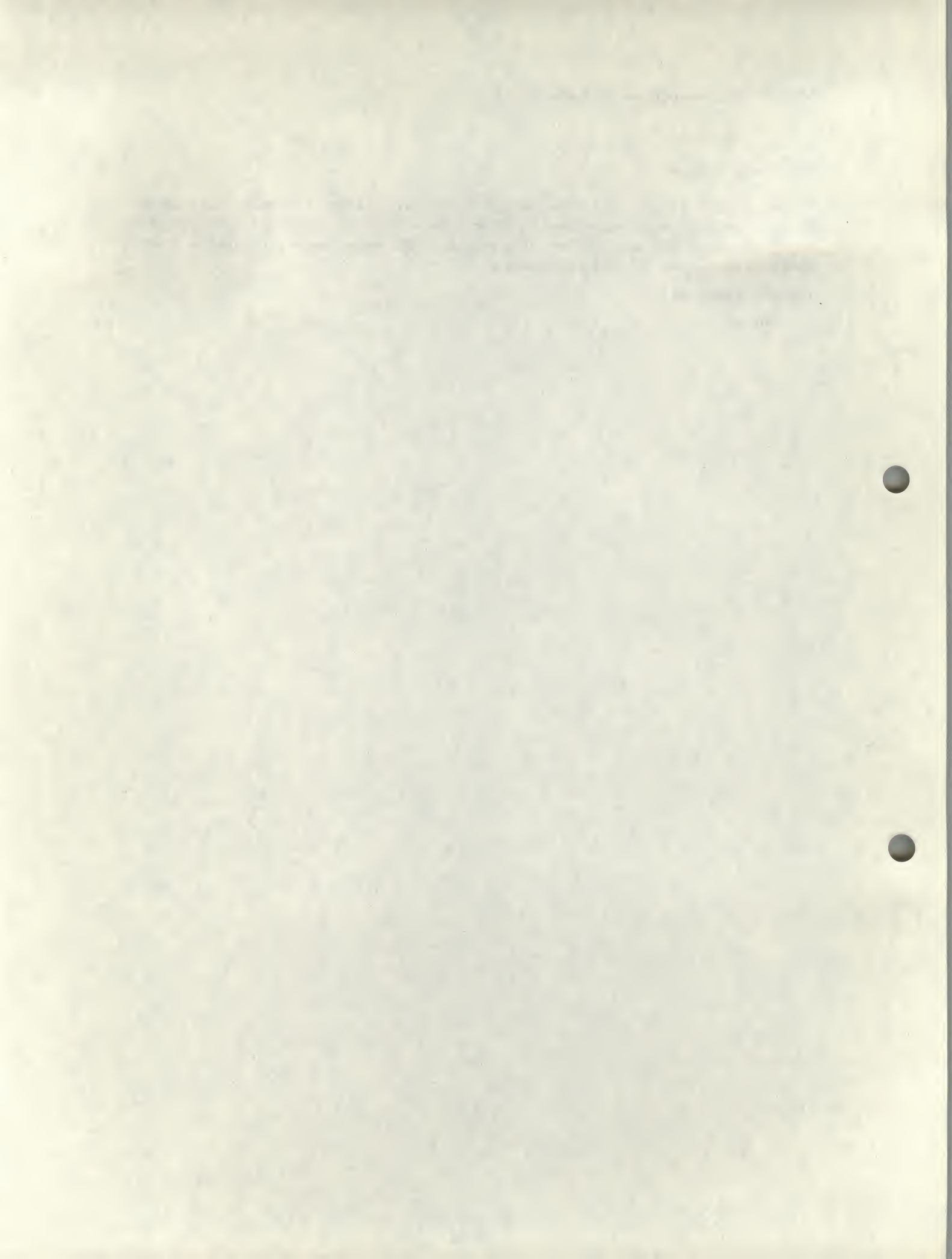
MACRO-11 Procedures With Pascal-2

The 'Endpr' Macro

The `endpr` macro marks the end of the procedure body. Only one `endpr` is allowed in each procedure. When the `endpr` is assembled, registers are restored, the variables and arguments are popped off the stack, and control is returned to the calling procedure. The `endpr` macro is designed to generate good code for popping the stack and returning.

The calling sequence is:

`endpr`



Type Definitions

In addition to the procedure definition macros described above, the PASMAL package defines some standard "types" and provides a set of three macros to simplify the definition of data structures. Each type is represented by its length in bytes.

The predefined types are:

<u>Type</u>	<u>Length</u>
char	1
boolean	1
scalar	1
integer	2
pointer	2
address	2
real	4
double	8
procpair	4

The type `procpair` is actually a record definition having two fields. This type is explained below.

The structure definition package consists of three macros:

<u>Name</u>	<u>Argument</u>	<u>Function</u>
<code>record</code>	<code>typename</code>	Begins the definition of a record type <code>typename</code> . The symbol <code>typename</code> will be set to the length of the record at the end of the definition. If the data type is <code>procpair</code> , the record is a procedure being passed as a parameter to a MACRO-11 routine.
<code>field</code>	<code>name</code> <code>size</code>	Defines a field in the record. The fields are allocated in ascending order, and any field with a length greater than 1 is allocated on a word boundary. Fields so defined are set equal to the offset of the field relative to the beginning of the structure.
<code>endrec</code>		Ends the definition of a record and assigns the total length to the <code>typename</code> given in the <code>record</code> macro.

For example, consider the following Pascal record definition:

```
prec = record
  Intf1: integer;
  Intf2: integer;
  Boolf1: boolean;
  Realf1: real;
end;
```

The equivalent code using the structure-definition macros is:

```
record prec ; prec = record
field intf1, integer ; intf1: integer;
field intf2, integer ; intf2: integer;
field boolf1, boolean ; boolf1: boolean;
field realf1, real ; realf1: real;
endrec ; end;
```

1890

Received of the Treasurer of the
Board of Education the sum of
\$100.00 for the year 1890

Witness my hand and seal this
1st day of January 1890
at New York City

John D. ...

John D. ...

John D. ...

John D. ...

John D. ...

John D. ...

MACRO-11 Procedures With Pascal-2

Later in the procedure, where the definition above occurs, we find:

```
var    local,prec      ; local: prec
```

And we would refer to field `intf2`, for example, as

```
mov    local+intf2(sp),r0
```

The type `procpa` is used in rare cases when you are passing a procedure as a parameter to a MACRO-11 routine. The definition is:

```
record  procpa
field   pp.proc,address      ; address of the procedure
field   pp.stat,address      ; address of the enclosing static link
endrec
```


Example

This example shows the coding of a MACRO-11 procedure for use with Pascal-2. The procedure chosen for the example is not one that would normally be coded in MACRO-11, but most such procedures are extremely dependent on the operating system. In fact, we begin with a version of the algorithm as it is coded in Pascal-2:

```
{ $nomain }

procedure CountOnes(N: integer;           { number to count bits in }
                   var Ones: integer;    { number of "one" bits }
                   var First: integer    { highest "one" bit });

external;

{ This is a procedure that counts the "one" bits in an integer and
  returns the number of ones in "Ones" and the highest bit found
  in "First". If no bits are set, "First" receives "-1".
  The procedure uses an extension of Pascal-2 that allows the
  signed number "N" to be treated as an unsigned number "TN".
}
procedure CountOnes;

var
  TN: 0..65535;           { local unsigned value of N }
  Bits: 0..16;           { bit count }

begin
  First := -1;
  Ones := 0;
  Bits := 0;
  TN := N;
  while TN <> 0 do begin
    if odd(TN) then begin
      Ones := Ones + 1;
      First := Bits;
    end;
    Bits := Bits + 1;
    TN := TN div 2;
  end;
end;
```

This simple procedure counts the number of bits set in an integer, checking whether the lowest bit is set, incrementing a counter, and terminating when there are no more bits set. The use of unsigned integers (TN, in the range 0..65535), avoids the shifting of the sign bit into the lower-order bits. (Unsigned integers are discussed in the Programmer's Guide and in the Language Specification.) This procedure (and many others that are often coded in low-level code) can be coded as a Pascal-2 procedure. But in many ways this procedure is typical of the sort of procedure you may code in MACRO-11:

- It performs a single function with simple internal logic.
- It is a generally useful form of the function, rather than a special use.
- It makes no reference to global variables. All data is passed as parameters.

The first example gives the most direct translation into MACRO-11, with all references to variables made directly to memory. It is quite possible to do the entire function in registers, with some saving

Received of the Hon. Secy. of the Navy
the sum of \$100.00 for the purpose of

the purchase of the following articles
to wit: 100 lbs. of sugar

100 lbs. of rice
100 lbs. of flour

100 lbs. of coffee

Witness my hand and seal this 1st day of
January 1862 at Washington

Very respectfully,
John A. Smith
Secretary of the Navy

MACRO-11 Procedures With Pascal-2

in code and execution time, but for the sake of the example we will not do this. We change the algorithm slightly to make use of the state of the condition code at the end of the loop. The use of a conditional branch at this point shortens execution time slightly at no cost in code size.

```
.title count

; This is a sample procedure that counts the number of bits
; set to one in a word "n" and also sets the variable "first"
; to the bit number of the highest bit set.
;
; This is used strictly as an example; some values that would
; normally be kept in registers are being kept in local variables
; or handled directly in memory for demonstration purposes.

proc countones ; procedure countones(
param n, integer ; n: integer;
param ones, address ; var ones: integer;
param first, address ; var first: integer);

; var
var bits, integer ; bits: integer; bit counter

begin ; begin
mov #-1, @first(sp) ; first := -1;
clr @ones(sp) ; ones := 0;
clr bits(sp) ; bits := 0;

tst n(sp) ; if n <> 0 then
beq 10$ ;
1$: ; repeat
bit #1, n(sp) ;
beq 2$ ; if odd(n) then begin
inc @ones(sp) ; ones := ones+1;
mov bits(sp), @first(sp)
; first := bits;
2$: ; end;
inc bits(sp) ; bits := bits + 1;
clc ;
ror n(sp) ; n := n div 2;
bne 1$ ; until r0 = 0;
10$: ;
endpr
.end
```

This procedure illustrates the use of parameters, local variables, and the **begin** and **endpr** macros. The local variable to hold *n* is not needed as there is no distinction made between signed and unsigned integers at the MACRO-11 level. The equivalent Pascal-2 code in the comments should make the MACRO code easy to follow.

In actual practice, local variables would be kept in registers, and the **save** and **restore** macros would be used to save and restore the registers used. The following version is an example of this

kind of code.

```
.title count

; This simple procedure counts the number of bits
; set to one in a word "n" and also sets the variable "first"
; to the bit number of the highest bit set.
;
; Functionally, this is the same procedure as "examp", except that
; it places local variables in registers whenever possible.
;

proc    countones      ; procedure countones(
param   n, integer     ;   n: integer;
param   ones, address  ;   var ones: integer;
param   first, address ;   var first: integer);

save    <r0,r1,r2,r3>

begin
mov     n(sp),r0        ; r0 := n;
mov     #-1,r1          ; r1 := -1; first
clr     r2              ; r2 := 0; ones
clr     r3              ; r3 := 0; bits

tst     r0              ; if r0 <> 0 then
beq     10$             ;
1$:                                           ; repeat
bit     #1,r0
beq     2$              ;   if odd(r0) then begin
inc     r2              ;   r2 := r2+1;
mov     r3,r1           ;   r1 := r3;
2$:                                           ;   end;
inc     r3              ;   r3 := r3 + 1;
clc                                           ;
ror     r0              ;   r0 := r0 shift 1;
bne     1$              ;   until r0 = 0;
10$:                                           ;
mov     r1,0first(sp)   ; first := r1;
mov     r2,0ones(sp)    ; ones := r2;
endpr
.end
```

In the Pascal program that invokes CountOnes, the following reference is made:

```
procedure CountOnes(N: integer;
var Ones: integer;
var First: integer);

external;
```


Placing PASMAL into the System Macro Library

If you often write Pascal programs that invoke MACRO-11 subroutines written using the PASM macro package, you might find it desirable to add the PASM macro package to your system macro library. This allows MACRO-11 programs to use PASM via the `.MCALL` assembler directive rather than by specifying PASM.MAC as a separate input file in the command line.

When you assemble a MACRO-11 subroutine, the assembler searches the system macro library (SY:SYSMAC.SML) to define macros requested with the **.MCALL** directive. The system macro library normally contains definitions of macros that call system services. You can easily add your own macro definitions to this library.

To add PASMAL to the system macro library, perform these steps:

1. Make a backup copy of your system macro library in case something goes wrong.
2. Using a text editor, create a file called P.MAC, which encloses PASMAC.MAC with a macro definition, as shown below. The definition creates a macro called PASMAC, which contains the entire contents of the file PASMAC.MAC. The macro definition redefines the macro PASMAC to be a null macro. This saves space and time in the assembler. The macro definition must be in upper case.

```
.MACRO PASMAL
:
: _____ contents of PASMAL.MAC
.MACRO PASMAL
.ENDM

.ENDM PASMAL
```

We offer two ways to create P.MAC. The first way is to create the skeleton macro, then insert the contents of PASMACH.MAC at the location shown above. The other possibility is to first copy PASMACH.MAC to P.MAC and then edit P.MAC, placing the skeleton macro directives around the contents of PASMACH.MAC.

3. Copy SYSMAC.SML to some other file, say SYSMAC.XXX.
4. Run the librarian to add the contents of P.MAC to SYSMAC.XXX. This produces SYSMAC.SML, the new system macro library.

.R LIBR
*SYSMAC.SML/M=SYSMAC.XXX,P.MAC

PASMAC.MAC is now a part of the system macro library. You can now use the **.MCALL** directive to define the PASMAC macros in your MACRO routines. This is illustrated below. The routine simply clears registers R0 and R1.

```
.title      test
.mcall     pasmac                ; Read the PASM MAC macro
pasmac     ; Define PASM MAC macro

proc       test                  ; Sample procedure
param     foo,integer
save      <r0,r1>
begin
clr       r0
clr       r1
endpr
.end
```


Placing PASMAC into the System Macro Library

Now you can assemble the routine without specifying the PASMAC.MAC source file on the **MACRO** command line.

.MACRO TEST=TEST

PROSE: A Text Formatter

Computerized text-processing tools such as text editors and formatters can ease the tedious preparation and editing of computer-oriented documentation. Instead of cutting, pasting, and retyping hard copy, you instruct the computer to insert changes, reformat and number pages, then reprint the document. PROSE, a text-formatting utility program, allows you to print any document in a variety of formats.

This guide describes the operation of PROSE, providing an overview of text-formatting procedures for PROSE and a detailed explanation of PROSE directives. The first-time user of PROSE can read this guide from beginning to end, using it as a tutorial while producing a first document. The more experienced reader may use it as a reference source. As an aid to both, this guide groups PROSE directives by function into four sections: controlling input, establishing format, indexing, and printing. The order of the directives in the guide reflects the order in which these directives might be applied to a text. The reader should have some basic knowledge of a text editor.

PROSE requires a small number of easily learned commands. Unlike some text-formatting programs, which use macro commands, variables, and other features usually associated with programming languages, PROSE does not overwhelm the user with complicated syntax. The text stands out, not the directives. This simplicity allows you to produce high-quality text with a minimum of effort.

Like many text formatters, PROSE will format text in pages, filling and justifying lines, placing titles and page numbers as needed. The following table shows some common features of text formatters that PROSE does and does not have.

Prose Can ...

Underline
Hyphenate words
Convert upper-case input
to mixed-case output
Produce a sorted index
Print selected pages

... And Cannot

Control photo-typesetting machines
Do graphics
Produce multi-column text
Store text and retrieve it later
Use tabs

PROSE may or may not be the tool for a given application.

THE HISTORY OF THE

First Part of the History of the

Second Part of the History of the

Third Part of the History of the

Fourth Part of the History of the

Fifth Part of the History of the

Sixth Part of the History of the

Seventh Part of the History of the

PROSE Basics

The basic units of any text-formatting system are the word, the line, and the paragraph. In PROSE, a word is defined as any non-blank string of characters, with a blank on either side. For the purposes of formatting, a punctuation character is part of the word next to it. A line consists of the number of words that PROSE can fill between margins. PROSE places as many words as possible into each output line, adding blanks to justify the lines to left and right margins.

Text formatting is largely filling and justifying, a process illustrated by the following example.*

Input to PROSE:

Eddie went to the ground floor cafeteria and got a sandwich and container of coffee, then went back to his office to work on the water bill survey. No one else was there; the others were still out on their regular lunch hour.

Why not? he asked himself. It took only ten minutes from start to finish: eight to find the code, one to decide how to do it, and one more to type the orders into the computer console. When he finished, the screen told him that the violation number had been removed.

A few minutes later his office door opened. It was his boss. He was back ten minutes early from his lunch hour.

"You're here," the boss said.

"That's right," Eddie said.

"Good," the boss said, leaving without bothering to close the door.

Output from PROSE:

Eddie went to the ground floor cafeteria and got a sandwich and container of coffee, then went back to his office to work on the water bill survey. No one else was there; the others were still out on their regular lunch hour.

Why not? he asked himself. It took only ten minutes from start to finish: eight to find the code, one to decide how to do it, and one more to type the orders into the computer console. When he finished, the screen told him that the violation number had been removed.

A few minutes later his office door opened. It was his boss. He was back ten minutes early from his lunch hour.

"You're here," the boss said.

"That's right," Eddie said.

"Good," the boss said, leaving without bothering to close the door.

When the user gives no special instructions, called directives, PROSE operates in the default mode as shown in the example above. In the default mode, PROSE automatically fills and justifies output lines, formatting the output into pages. Directives instruct PROSE to do anything more sophisticated. If the user doesn't enter certain directives, PROSE supplies their default forms.

* Text examples in this guide have been excerpted from *The Programmer* by Bruce Jackson. Copyright © 1979 by Bruce Jackson. Reprinted by permission of Doubleday & Company, Inc.

The first of the three main parts of the book is devoted to a general survey of the history of the world from the beginning of time to the present day. The second part is devoted to a detailed account of the history of the United States from the first settlement to the present day. The third part is devoted to a detailed account of the history of the British Empire from the first settlement to the present day.

The first of the three main parts of the book is devoted to a general survey of the history of the world from the beginning of time to the present day. The second part is devoted to a detailed account of the history of the United States from the first settlement to the present day. The third part is devoted to a detailed account of the history of the British Empire from the first settlement to the present day.

The first of the three main parts of the book is devoted to a general survey of the history of the world from the beginning of time to the present day. The second part is devoted to a detailed account of the history of the United States from the first settlement to the present day. The third part is devoted to a detailed account of the history of the British Empire from the first settlement to the present day.

The first of the three main parts of the book is devoted to a general survey of the history of the world from the beginning of time to the present day. The second part is devoted to a detailed account of the history of the United States from the first settlement to the present day. The third part is devoted to a detailed account of the history of the British Empire from the first settlement to the present day.

PROSE: A Text Formatter

Structure of Directive Lines

In general, a directive line has three components: the escape character, the directive name, and the parameter for its application. Most PROSE directive lines take one of three forms:

```
.directive name  
.directive name integer  
.directive name( parameter )
```

The **directive escape character** is placed in the first column of an input line to indicate that at least one directive follows. The period (.) is the default escape character because it seems unlikely that anyone would want to type a period in the first column of a line of text. The default can be changed with the **INPUT** directive (see "Controlling Input to PROSE").

After the escape character comes the name of the directive that PROSE is to execute. The *directive name* can be abbreviated to three letters (in fact, PROSE only examines the first three). Examples in this guide show directives typed in upper case, but PROSE accepts both cases.

The directive name may or may not be followed by a *parameter*. The **BREAK** directive, for example, doesn't require a parameter. If a necessary parameter is omitted, PROSE supplies a default value for that parameter. The default values that PROSE uses are listed in a table of options under each directive.

A parameter can be one of three types:

- Text on the remainder of the directive line.
- An integer.
- Any specific options enclosed in parentheses, consisting of other directive names, integers, or keywords defined by the directive itself.

In text-processing systems such as PROSE, a **keyword** (also called a descriptor) categorizes or indexes information. Many PROSE directives use special letters or characters to express the options assigned, as do the **INPUT** or **FORM** directives. In directives such as **RESET(MARGIN)**, the keyword is the name of the directive to be changed. The summary directive table in Appendix B indicates the parameter type that each directive may take.

Numeric values used as a parameter or part of a keyword may be either an explicit positive integer or a relative value. A relative value, specified by a plus or minus sign before the integer, indicates that the old value should be increased or decreased by the amount of the integer. For example, if the left margin is set to 10 and the right margin to 70, you could use a relative values in the directive

```
.MARGIN( L+5 R-5 )
```

to squeeze the margins together by 5 characters on each side.

Placement of Directives

Directive lines are usually separated from lines of text (see the following sample file). Several directives can be typed on the same line, provided that they are separated by the directive escape character, as follows.

```
.BREAK.SKIP 2.MARGIN( L5 R65 )
```

Some directives take the remainder of the line as their parameter, so no other directives can follow these (e.g., **COMMENT** directive). The following sample shows the placement of PROSE commands in an input file.

Input to PROSE:

```
.COMMENT This example makes very primitive use of directives,
.COMMENT but it will produce exemplary text.
.MARGIN( L10 R60)
.INDENT 2
Eddie ordered that the tax roll and Yellow Pages tapes be returned
to storage. A few seconds later the video screen told him they had
been returned to their appropriate storage locations.
.BREAK.INDENT 2
Eddie smiled at the screen. He loved the computers. They would do
exactly what you told them to do and they would never lie to you.
Two inhuman characteristics. People who complained about the inhumanity
of computers were right. They didn't know how to care or betray.
.BREAK.INDENT 2
The next operation was more complicated. He had prepared for
it some time before, and the preparation had required many
separate inquiries.
```

Output from PROSE:

Eddie ordered that the tax roll and Yellow Pages tapes be returned to storage. A few seconds later the video screen told him they had been returned to their appropriate storage locations.

Eddie smiled at the screen. He loved the computers. They would do exactly what you told them to do and they would never lie to you. Two inhuman characteristics. People who complained about the inhumanity of computers were right. They didn't know how to care or betray.

The next operation was more complicated. He had prepared for it some time before, and the preparation had required many separate inquiries.

For more sophisticated examples, see "Appendix A: Examples of PROSE Directives in Text."

A long directive may extend beyond one line. A continued line is indicated by a **continuation character**, a plus sign '+' placed in column one. The following example shows suggested placement of the continuation character:

```
.FORM( [ /// L58 // #73 'PAGE' p /// ]
+      [ /// L58 //      'PAGE' p /// ] )
```

Generally, directives are placed at the beginning of the input file or at the point in the text where the directive takes effect. Most directives either control the functions of PROSE or set general format guidelines for the document. These are placed at the beginning of the text and their operations are applied throughout, unless temporary changes are made. The **FORM** directive, for example, establishes page format for the rest of the text, but the number of lines per page can be adjusted by an option of the **PARAGRAPH** directive. Directives that apply only to a particular line, such as **INDENT**, **BREAK**, and **COMMENT** in the sample above, are placed wherever necessary throughout the text.

January 1, 1900

My dear Mr. [Name],
I have just received your letter of the 28th inst. and am
glad to hear that you are well. I am
also well and hope this letter finds you
the same. I have been thinking of you
often and wondering how you are getting on.
I have been very busy lately but
will try to write you more often.

I have been thinking of you
often and wondering how you are
getting on. I have been very busy
lately but will try to write you
more often. I hope this letter
finds you well. I am also well
and hope this letter finds you the
same.

I have been thinking of you
often and wondering how you are
getting on. I have been very busy
lately but will try to write you
more often.

I have been thinking of you
often and wondering how you are
getting on. I have been very busy
lately but will try to write you
more often.

PROSE: A Text Formatter

Running the PROSE Program

No actual formatting takes place until the input file, containing text and directive lines, is submitted to PROSE for processing. You create the input file with your system's text editor. PROSE places the formatted text in an output file for submission to a printer or for display on a terminal screen.

To format the input file, invoke PROSE as shown:

```
.R PROSE  
*output-file = input-file
```

input-file:

The PROSE source file(s). Multiple input files are read and concatenated from left to right. The default extension for input files is .PRS.

output-file:

The formatted PROSE file. If the output file and the '=' separator are omitted from the command line, the output file will take the name of the last input file and the default extension .DOC. Default output files are placed in the default directory.

The output file may then be printed. The formatting and printing operations may be merged by means of header files.

Header Files

Certain directives nearly always appear at the head of any input file. If all of your documents use these directives in the same form, you can set up a header file rather than type them in each document's input file. Header files also provide you with an easy way to choose among various forms or output devices.

As a general practice, we recommend that you set up each PROSE text without OUTPUT or FORM directives. Instead, keep these directives in another file that you will use as the first input file, or "header file." For example, you may wish to create a header file for output to a video terminal and another for output to the line printer.

If your header files are stored on the system device, you would use this command to prepare the document for the terminal (assuming the header file is named VT100.PRS):

```
.R PROSE  
*DOCNAM = SY:VT100,DOCNAM
```

and this command to prepare the document for the line printer (assuming the header file is named PRINTR.PRS):

```
.R PROSE  
*DOCNAM = SY:PRINTR,DOCNAM
```

PROSE prints the output file according to directives in the header file. See "Page Format" and "Specifying Output Devices" for the functions of the individual directives included in the header file.

Controlling Input to PROSE

The directives in this section control the input to the PROSE program. Generally, they are placed at the beginning of the input file for a document or in a header file to be used for all documents. You can set and change them as needed throughout the text.

INPUT Directive

The **INPUT** directive tells PROSE how to interpret certain control characters in the input file and sets the maximum length for input lines.

The following table summarizes the options for its parameter.

<u>Key Letter</u>	<u>Meaning</u>	<u>Type</u>	<u>Default</u>
B	Explicit blank character	character	nul
H	Hyphenation character	character	nul
C	Case-shift character	character	nul
U	Underline character	character	nul
D	Directive escape character	character	.
W	Input width	number	150
K	Keep	number	next

The options, which can be given in any order, consist of a key letter followed by a value. Unless the user specifies both the key letter and a value, the default value is assigned when PROSE begins processing. A value in the parameter changes **only** when a new value is given. No **INPUT** option uses relative values.

- B:** The **explicit blank character** indicates a blank that PROSE should treat as if it were a character. With the cross-hatch '#' specified as the explicit blank, the following example shows how two words separated by an explicit blank will never be split from one line to the next. PROSE will never fill blanks between the words to justify a line.

```
.INPUT( B# )
```

```
...someone like the imaginary Dr.#Conrad and...
```

- H:** The **hyphenation character** defines hyphenation points within words. Sometimes a long word will cause many blanks to be inserted to justify the preceding line. PROSE will hyphenate such a word if you have defined the syllable boundaries within it. Of course, not all the syllable boundaries need be specified, only those at which you want PROSE to be able to split a word. For example, if the hyphenation character is the slash '/', you may type "syncopation" as **syn/co/pa/tion**. PROSE will insert a hyphen '-' only when the characters on both sides of the hyphenation point are letters. This restriction allows you to type "hyper-active" as **hyper-/active**, and PROSE will split the word if necessary, without adding a superfluous hyphen. If PROSE is forced to insert blanks beyond a certain threshold set by the **OPTION** directive, PROSE will issue an error message on the line that needs hyphenation characters.
- C:** To produce mixed-case output from upper-case-only input, you must specify a **case-shift character** in the **INPUT** directive parameter, causing PROSE to automatically shift all upper-case letters to lower case. To preserve certain upper-case letters, such as initial capitals for names and sentences, you can surround the letter or letters with case-shift characters. PROSE shifts to upper case for all characters between case-shift characters. "Stuttering" is another way to designate capitals among upper-case-only input. Since most upper-case letters are at the beginning of a word (following a blank), you use two letters to indicate a single capital. Words that already begin with a double letter will produce a single capitalized letter unless you put two case-shift characters before the word.

My dear Mr. [Name],

I have just received your letter of the 15th inst. and am glad to hear from you. I am well and hope this finds you the same.

I am sorry to hear that you are not well. I hope you will soon be able to return to your duties.

I am sure you will find the enclosed of interest.

I am, dear Mr. [Name], very respectfully,
Your obedient servant,
[Signature]

[Address]

I am, dear Mr. [Name], very respectfully,
Your obedient servant,
[Signature]

[Address]

I am, dear Mr. [Name], very respectfully,
Your obedient servant,
[Signature]

[Address]

I am, dear Mr. [Name], very respectfully,
Your obedient servant,
[Signature]

PROSE: A Text Formatter

<u>Input to Pr:</u>	<u>Output from Pr:</u>
^^LLAMA	llama
^^OOPS	oops
LLLAMA	Llama
OOOPS	Oops

The following example demonstrates both ways of producing mixed-case output from upper-case-only input. The case-shift character is easier to use for long strings, such as example programs, that are to be capitalized. Stuttering is easier when you want to capitalize a single character, such as the first word of a sentence. You can use both methods in the same text as shown below.

Input to PROSE:

```
.INPUT(C~)
HHE HAD EIGHTEEN MINUTES, PLUS THE LOCAL CONNECTION. TTWENTY-ONE
MINUTES. AA WORLD OF TIME ON A COMPUTER. HHE WAS READY WITH HIS
QUESTIONS.
^WHAT IS CODE FOR PROGRAM?^
^COMPUTER BANDIT.^
^WHAT IS KNOWN ABOUT COMPUTER BANDIT?^
TTHE SCREEN RAPIDLY FILLED WITH THE DATES AND AMOUNTS OF HIS
AAMERICAN EEXPRESS THEFTS, SOME OF HIS RECENT INFORMATION SCANS,
THE REPORTS TO THE NEWSPAPERS IN NNEW YYORK.
```

Output from PROSE:

```
He had eighteen minutes, plus the local connection. Twenty-one
minutes. A world of time on a computer. He was ready with his
questions.
WHAT IS CODE FOR PROGRAM?
COMPUTER BANDIT.
WHAT IS KNOWN ABOUT COMPUTER BANDIT?
The screen rapidly filled with the dates and amounts of his American
Express thefts, some of his recent information scans, the reports to
the newspapers in New York.
```

For conversion of mixed-case input to upper-case-only output, see the **OPTION** directive.

- U: Text surrounded by the **underline** character will be underlined. Blanks are not underlined, but explicit blanks are.
- D: The **directive escape character** is placed in the first column of an input line to flag it as a directive. Use this option only to define a directive escape character other than the period.
- W: The input width **W** specifies the number of characters to be read from each input line. Users will only need to change the input width for special jobs.
- K: The **keep** option explicitly specifies the keep buffer to be used to store the new input options. By default, PROSE uses the numerically next buffer. (See "Changing The Format Control Directives" for detailed explanation of the use of **keep buffers** in PROSE directives.)

OPTION Directive

The **OPTION** directive gathers together miscellaneous options that affect the filling and justifying PROSE does during text formatting. These options are summarized in the following table. Key letters are followed by a switch symbol (+/-) or an integer, as shown in the default column. For the switch-type options, the plus sign '+' means on and the minus sign '-' means off.

My dear Mr. [Name],

I have just received your letter of the 10th inst.

and am glad to hear from you.

I am well and hope this finds you the same.

I have not yet had time to write you more fully.

I am, dear Mr. [Name], very respectfully,

Your obedient servant,

[Signature]

[Address]

[City, State]

[Post Office]

Controlling Input to PROSE

<u>Key Letter</u>	<u>Meaning</u>	<u>Default</u>
E	Print error messages	+
J	Justification limit	3
F	Fill output lines	+
L	Left justify	+
R	Right justify	+
S	Spacing	1
M	Multiple blanks	+
P	Two blanks after periods	+
U	Shift to upper case	-
K	Keep	next

As processing begins, PROSE assigns the default value for each option without a specified value. A parameter value changes only when a specification is given. No option uses relative values.

- E: Error messages appear in the formatted text of the main output files at the approximate location of the errors. Error messages are suppressed when this option is off (E-).
- J: PROSE inserts blanks as needed to justify the left and right margins of an output line. The **justification limit** controls the point at which PROSE will attempt to hyphenate a word. If, for instance, the justification limit is set at 3, then the hyphenation process will be invoked when PROSE has to insert three blanks between adjacent words on a line. If hyphenation is not possible, or PROSE is not able to bring the number of inserted blanks below the limit, an error message is printed for the line(s).

NOTE

Settings for options E and J can be varied according to the draft you are working on. Setting J to an arbitrarily high number (e.g., 20) and turning off E helps you to avoid hyphenation errors until you are ready to deal with them, usually in the later stages of document preparation.

- F: Output lines are automatically filled and justified as described in the "PROSE Basics" section. If the **fill** option is off, PROSE will print the input lines as they are, without reformatting to fill the output lines. In effect, a justification break is done after each input line. Option F- is most useful for literal text, such as program examples, where spacing between words must be exactly as typed.
- L: The **left** and **right justify** switches work together to determine the justification to be done. If both options are on, output lines are justified to both the left and right margins. If both options are off, the lines are centered between the two margins. If one is on and the other is off, one margin (either left or right) will be straight and the other ragged. The following examples demonstrate the output from the four combinations.

Output from .OPTION(L+ R+) :

Eddie did four more operations, three of them involving county and city payroll checks, all of which were handled by the same computer.

Output from .OPTION(L- R-) :

He gave an across-the-board raise of fifty dollars per check to all teachers in the ghetto schools. He deducted an equivalent amount from the checks of the city's highly paid political appointees.

Output from .OPTION(L+ R-) :

Then he erased the tapes for outstanding private residential water bills. People, he decided, shouldn't have to pay for a drink of water or to be able to flush ... a toilet.

[Faint, illegible text, likely bleed-through from the reverse side of the page]

PROSE: A Text Formatter

Output from .OPTION(L- R+) :

The final operation was one he hadn't thought of earlier; it had come to him during the night's work. It was easy enough with the information he now had.

- S: The **spacing** option 2 generates double-spaced output; the spacing option 3 generates triple-spaced output. By default, text is single-spaced.
- M: If the **multiple blanks** option is on (M+), multiple blanks in the input file are considered to be significant. That is, if several blanks are placed between two words in the input file, at least that many will appear in the output file; PROSE may add blanks during justification. If the option is off, multiple blanks will be treated as a single blank.
- P: The **2 blanks after periods** option places at least two blanks after every period. PROSE will not add blanks before justifying if two are already present. This makes for consistent spacing in the final copy even if you are not careful about typing 2 spaces after sentence periods in the original. Three or more blanks after a period are treated as multiple blanks.
- U: For output devices that cannot process mixed-case files, the **shift to upper case** option shifts all lower-case letters to upper-case letters. This option is also useful for printing an entire passage or example, such as a sample program, all in upper case. For conversion of upper-case-only input to mixed-case output, see the **INPUT** directive.
- K: The **keep** option explicitly specifies the keep buffer to be used to store the new options. By default, PROSE uses the numerically next buffer. (See "Changing Format Within the Text" for use.)

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE RESEARCH GROUP ON
THE CHEMISTRY OF THE CARBON
AND SILICON COMPOUNDS

BY
J. H. HARRIS, JR.
AND
J. H. HARRIS, JR.

CHICAGO, ILLINOIS
1954

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE RESEARCH GROUP ON
THE CHEMISTRY OF THE CARBON
AND SILICON COMPOUNDS

BY
J. H. HARRIS, JR.
AND
J. H. HARRIS, JR.

CHICAGO, ILLINOIS
1954

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE RESEARCH GROUP ON
THE CHEMISTRY OF THE CARBON
AND SILICON COMPOUNDS

BY
J. H. HARRIS, JR.
AND
J. H. HARRIS, JR.

CHICAGO, ILLINOIS
1954

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE RESEARCH GROUP ON
THE CHEMISTRY OF THE CARBON
AND SILICON COMPOUNDS

BY
J. H. HARRIS, JR.
AND
J. H. HARRIS, JR.

CHICAGO, ILLINOIS
1954

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY

REPORT OF THE RESEARCH GROUP ON
THE CHEMISTRY OF THE CARBON
AND SILICON COMPOUNDS

BY
J. H. HARRIS, JR.
AND
J. H. HARRIS, JR.

CHICAGO, ILLINOIS
1954

Setting Up the Document's Format

This section explains the use of directives to format text. These directives specify page format, margins, paragraphing conventions, justification breaks, and blank or comment lines.

Page Format

The **FORM** directive defines the page format, including insertion of titles, date/time, blank lines, page numbers, and other textual items at the top or bottom of the page. The **FORM** directive works with the **COUNT**, **TITLE**, and **SUBTITLE** directives. The **PAGE** and **PARAGRAPH** directives can override the page-break function of the **FORM** directive.

FORM Directive

The **FORM** directive can produce a variety of page formats, depending upon the options specified in its parameter. The table below contains the available **FORM** options; the following paragraphs explain their use.

<u>Key Char</u>	<u>Meaning</u>	<u>Default Field Width</u>
[	Define top of page	-none-
]	Define bottom of page	-none-
#n	Tab forward or backward to absolute column n	-none-
S	Subtitle	its length
T	Main title	its length
Ln	Fill in n lines of running text on the page	-none-
/	Print an end of line (by itself, a blank line)	-none-
/n	Print n ends of lines	-none-
Pf	Current page number, f selects the form:	3
	N or n Arabic numerals (default)	[The field width will be expanded if needed]
	L Upper-case letter	
	l Lower-case letter	
	R Upper-case Roman numerals	
	r Lower-case Roman numerals	-none-
'...'	Print material within quotation marks as literal text	
C	24-hour clock as hh.mm.ss (e.g. 15.37.58)	
D	Raw date as yy/mm/dd (e.g. 82/02/13)	
E	Nice date as dd Mmm yy (e.g. 13 Feb 82)	
W	Wall clock as hh:mm PM or hh:mm AM (e.g. 3:37 AM)	8

If the **FORM** directive is omitted completely, PROSE uses the default form:

```
.FORM( [ // T #62 E /// L54 /// #33 '- ' PN:1 ' -' //// ] )
```

The sequence of options within parentheses corresponds to the format of the page from top to bottom. The **FORM** directive builds text lines from left to right, starting in the first printable column unless a tabbing specification (#n) starts text at a specific column.

FORM directive parameters generally begin and end with the definition characters for a top-of-page and bottom-of-page. The top-of-page definition '[' has several uses. You can direct PROSE to send a page eject to the output device when it reaches the top of a page. Also, you can request a pause at the top of each page to allow you to change paper on the printer (see information on the **OUTPUT** directive in "Printing the Document"). At the end of the document, PROSE signals one last page eject and continues to interpret the **FORM** specification until it reaches another top-of-page. This ensures the execution of any commands specified for the bottom of the last page, such as a page

1880

My dear Mr. [Name]

I have just received your letter of the 10th inst.

and am glad to hear from you.

I am well and hope this finds you the same.

I have not much news to write at present.

I am, dear Mr. [Name], very respectfully,

Your obedient servant,

[Signature]

[Address]

PROSE: A Text Formatter

number. PROSE increments the page number at the bottom-of-page character ']'. So, if you print the page number both before and after the bottom-of-page definition, you will get different numbers.

To print slightly differing formats for facing pages, specify a format for each page between a pair of page definition characters. For example, this form directive prints the page number at the bottom right of odd numbered pages and at the bottom left of even pages.

```
.FORM ( [ // T #62 E /// L56 // #63 'PAGE' P /// ]  
+      [ // T #62 E /// L56 //      'PAGE' P /// ] )
```

Appendix A contains another example of the **FORM** directive used to print facing pages.

Page length is determined by the specification for the number of lines. PROSE breaks pages at the number of lines set by the **FORM** directive's **Ln** specification, unless the **PAGE** directive or the optional automatic page eject for the **PARAGRAPH** directive is used (see "Page Breaks"). If the **Ln** specification is omitted entirely, PROSE supplies the default value of 54 lines per page. If the **FORM** directive parameter contains a key letter **L** without a value, no special page formatting is done. Page length will be infinite, which is useful for working with documents on terminals, where pages are irrelevant. In this mode, a **PAGE** directive with no parameter will put 5 blank lines between sections of text.

Titles, subtitles, page numbers, and dates are placed in fields at the top or bottom of the page. Although default values are sufficient for most situations, the field width can be set to a particular value by placing a colon and the value after the key letter. For example, **T:30** prints the title in a field of 30 characters. Specified field widths are sometimes useful for truncating long titles. PROSE fills the field from right to left. The tabbing specification **#n** places the field horizontally on the page.

The **FORM** argument is re-scanned as each page of output is produced, so that any change in a title buffer made with the **TITLE** or **SUBTITLE** directive will insert the new title or subtitle on the next page. The **TITLE** directive enters the remainder of the line into the main title buffer. The **FORM** directive uses the contents of the title buffer to print a title on the page as specified. The **SUBTITLE** directive enters the remainder of the line into the subtitle buffer, to be used by the **FORM** directive to print a subtitle on the page as specified. See examples in Appendix A.

PROSE adds a blank line for each '/' mark in the **FORM** directive. These blanks are placed uniformly on each page, in the relative position that they appear in the directive, usually at the top and bottom, between the body of the text and the title and page number. The alternative **/n** allows a shorter expression of numerous blank lines; either form may be used.

Page numbers are incremented by the page counter and placed on the page by the **Pf** option of the **FORM** directive. The **COUNT** directive sets the page counter. The integer in the **COUNT** parameter can be a relative value; for example, **.COUNT +1** increments the page number by one. By default, the page counter sets the page number to 1.

The **literal text** option allows you to add such touches as hyphens surrounding the page number (see default **FORM** directive) or other text that must appear exactly as typed. For example, suppose a press release required the word "more" with parentheses around it at the bottom of each page. You could use the literal text specification as follows:

```
.FORM( [ T /// L54 /// #28 '(more)' /// ] )
```

PROSE users may choose between four styles of dates. The date is placed on the page in much the same manner as a title.

Page Breaks

The **PAGE** directive signals a page eject when fewer than the specified number of lines remain on the current page. If no parameter is given, the **PAGE** directive does an unconditional page eject. The **PARAGRAPH** directive's **automatic page eject** includes the page-break function in the paragraph format for the document. **.PAGE 3** and **.PAR(P3)** are equivalent, except that **.PAGE 3** must be

10/1/72

Page 1 of 2

The following information was obtained from the records of the
Department of the Interior, Bureau of Land Management, for the
year ending December 31, 1971. The information is presented in
the following order: (1) Total land area, (2) Land under
various uses, (3) Land owned by the Federal Government, (4)
Land owned by the State of California, (5) Land owned by
private individuals, and (6) Land owned by other entities.
The total land area in California is 158,333,436 acres.
Land under various uses is as follows: (1) Forest land,
10,123,456 acres; (2) Rangeland, 45,678,901 acres; (3)
Cultivated land, 12,345,678 acres; (4) Urban and
developed land, 2,345,678 acres; (5) Barren land,
10,987,654 acres; (6) Water, 10,000,000 acres.
Land owned by the Federal Government is 10,123,456 acres.
Land owned by the State of California is 45,678,901 acres.
Land owned by private individuals is 12,345,678 acres.
Land owned by other entities is 2,345,678 acres.

PARAGRAPH directive's automatic page eject includes the page-break function in the paragraph format for the document. **.PAGE 3** and **.PAR(P3)** are equivalent, except that **.PAGE 3** must be explicitly placed by the user, while PROSE executes **.PARAGRAPH (P3)** wherever indicated by the paragraph flag character.

Margins

The **MARGIN** directive sets the left and right margins for filling and justifying. The value for **left margin** indicates the column in which the line of text begins; the **right margin** value is the column number of the last printed character. Thus, subtracting the left margin from the right margin gives the number of columns for printed text.

The options, which may be given in any order, consist of a key letter followed by a value. The next table lists the key letter for each option.

<u>Key Letter</u>	<u>Meaning</u>	<u>Type</u>	<u>Default</u>	<u>Relative</u>
L	Left margin	integer	0	yes
R	Right margin	integer	70	yes
K	Keep	integer	next	no

Margins are set before text processing begins. PROSE assigns default values of **L0 R70** if no **MARGIN** directive is used or if the directive is given without a parameter. A value changes only when a new specification is given. The **keep** option explicitly specifies the keep buffer to be used to store the new margins. By default, PROSE uses the numerically next buffer.

The **INDENT** directive moves the next line of text to the right of the page by the given number of spaces. When the directive is used without a parameter, the default value is 5. The **UNDENT** directive moves the next line a certain number of spaces to the left. (The **undent** is sometimes known by the name "outdent" or "hanging indent.") If the given parameter would undent the text past the leftmost column of the printed page, the directive undents only to the leftmost printable column. If no parameter is given, the default undents to the leftmost printable column.

Paragraphs

Although you may use any justification break methods to distinguish between one paragraph and the next, the **PARAGRAPH** directive provides a more versatile method of creating paragraphs.

Placed at the beginning of the text, the directive sets the general form of paragraphs and specifies a paragraph flag character. Many **PARAGRAPH** options take the place of other directives, so the directive is a powerful tool.

When the paragraph flag character is placed in the first column of a text line to signal a new paragraph, PROSE takes any of the following actions that are specified by the parameter.

<u>Key Letter</u>	<u>Meaning</u>	<u>Type</u>	<u>Default</u>
F	Paragraph character	character	nul
I	Automatic indent	number	0
U	Automatic undent	number	0
N	Number generator		-none-
P	Automatic page eject	number	0
S	Automatic skip	number	0
K	Keep	number	next

Handwritten text at the top of the page, possibly a title or header.

Handwritten text block, likely the first paragraph of the document.

Handwritten text block, likely the second paragraph of the document.

Handwritten text block, likely the third paragraph of the document.

Handwritten text block, likely the fourth paragraph of the document.

Handwritten text block, likely the fifth paragraph of the document.

Handwritten text block, likely the sixth paragraph of the document.

Handwritten text at the bottom of the page, possibly a signature or footer.

PROSE: A Text Formatter

When it begins processing, PROSE assigns the default value for each option in the **PARAGRAPH** directive parameter. If the input file contains no **PARAGRAPH** directive, or if an option is not specified, the default value is used. No **PARAGRAPH** option uses relative values.

By manipulating the options in the parameter, you may direct PROSE to take any of the following actions for paragraphs.

- F:** The **paragraph flag character** invokes a collection of paragraphing actions when it appears in the first column of an input line. Note that this character must be set in the first **PARAGRAPH** directive, or no other options apply. As the only specified option, the paragraph flag character signals a justification break.
- I:** The **automatic indent** or **automatic undent** applies to the first line of the paragraph and moves the line left or right a given number of spaces. If the number generator is used, the indent or undent is applied after the number is generated (see the example using both options below).
- N:** The **number generator** produces a new number (or letter) for each occurrence of the paragraph flag character. PROSE inserts the number in lieu of the paragraph flag character when the line is formatted, so you must put a space between the paragraph flag character and the text line, if you want one to appear in the output. The number generator is initialized to 1 each time new paragraph settings go into effect. Resumption of an old setting also resumes the old numbering. The number generator's keyword contains these fields (spaces not allowed):

N numeric-field field-width

The key characters for *numeric-field* are:

-blank-	No numbering
N or n	Arabic numerals
L	Upper-case letter
l	Lower-case letter
R	Upper-case Roman
r	Lower-case Roman

The field width for the numeric field, expressed as an integer, is expanded if necessary. If, for example, you want an Arabic numeral with three spaces left for the numeral, the keyword is **Nn3**.

The next input and output examples illustrate one style of numbered, undented paragraph created with the automatic undent and number generator options. Note that the margin adjustment places the paragraph number in the leftmost column of the printed page.

Input to PROSE:

```
.MARGIN( L10 )
.PARAGRAPH( F& Nn1 U5 )
& Eddie worried about that one for a while, then came up with
a very simple answer: he would ask the computers if they had
any such self-inspection instructions in their programs. It
would become part of his regular greeting: HELLO. HOW ARE YOU
AND ARE YOU PROGRAMMED TO TRAP ME?
```

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

Output from PROSE:

1 Eddie worried about that one for a while, then came up with a very simple answer: he would ask the computers if they had any such self-inspection instructions in their programs. It would become part of his regular greeting: HELLO. HOW ARE YOU AND ARE YOU PROGRAMMED TO TRAP ME?

- P: The automatic page eject simulates the effect of the PAGE directive. For instance, the directive .PAR(P4) causes PROSE to eject a page if fewer than four lines of the paragraph are left at the bottom of the page. The command is applied after the automatic skip.
- S: The automatic skip functions the same as a SKIP directive, placing a blank line before the first line of the paragraph.
- K: The keep option explicitly specifies the keep buffer to be used to store the new paragraph options. By default, PROSE uses the numerically next buffer.

Values and options can be changed for particular paragraphs or sections of the document, as explained in "Changing Format Within the Text."

Comments

Using the COMMENT directive, you can include information in the source of a document that will not be printed in the formatted copy. As shown in the examples in Appendix A, PROSE treats the remainder of the line as a comment and ignores it.

Changing Format Within the Text

To make the fullest use of PROSE, the user must manipulate such options as blank characters, spacing, margins, or page breaks within the text. The BREAK and SKIP directives allow you to interrupt the established formatting process.

At certain points, you may need to switch formats for specific situations, such as example programs or blocked quotations. OPTION, MARGIN, and PARAGRAPH settings may change frequently, but the number of different settings will probably be predictable and few. Depending upon the number, the variety, and the frequency of changes the text requires, the following techniques can help you to enter new directives or restore previously used options.

Breaking and Skipping Lines

One of the simplest and most frequently used instructions, a justification break causes PROSE to stop filling the current output line and print it without justifying. A line break can be indicated in many ways. Text can be separated (broken) by one or more blank lines inserted in the text,

PROSE: A Text Formatter

by leading blanks typed on an input line (a paragraph indentation), or by the **BREAK** directive. The following example illustrates these three methods.

Input to PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
.BREAK
"Right," Barstow said.
.BREAK
"It's like Battleship," Purvey said, smiling at Barstow
and the IBM man.

"You understand perfectly," the IBM man said, "That's
exactly what it is."
```

Output from PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
"Right," Barstow said.
"It's like Battleship," Purvey said, smiling at Barstow and the IBM
man.

"You understand perfectly," the IBM man said, "That's exactly what it
is."
```

With any of these methods, you will only direct PROSE to do a justification break. PROSE will not skip lines or indent unless you explicitly enter blank lines or indentions in the input file.

The **SKIP** directive prints blank lines within the text by skipping a certain number of output lines. SKIP will not print blank lines at the top of a page, unless you enter at least one actual blank line before the **SKIP** directive. The default value of the **SKIP** directive is 5 lines.

Keep Buffers

The **keep buffer** is a simple way to change directives that control input or format. Each time a change in one of these directives is processed, PROSE saves the new values in a **keep buffer**. Ten keep buffers (0 through 9) are associated with each directive. You may use a keep parameter to specify the buffer to be used; if no buffer is specified, the values are saved in the numerically next buffer. To recall a previously used value, you enter the directive with the number of the keep buffer as the parameter.

For example, suppose that a double-spaced text has a number of paragraph-length quotations, which are to be typed as single-spaced blocks indented ten spaces from each margin. Using the keep option in the parameters for the **INPUT**, **MARGIN**, and **PARAGRAPH** directives, you could store the format specifications for each situation in two different keep buffers by entering these directives at the beginning of the input file:

```
.OPTION(K1 S2).MARGIN(K1 L10 R70).PARAGRAPH( K1 F& I2 S1 )
```

Then you enter these directives for the new format

```
.OPTION(K2 S1).MARGIN(K2 L20 R60).PARAGRAPH( K2 F& IO SO )
```

before the first blocked paragraph. To resume the standard paragraph format, you then enter the directive names with the number of the keep buffer. The input and output files would look like this:

Dear Sir,

I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the above matter.

I am sorry to hear that you are not satisfied with the result of the examination. I have been very anxious to see that all the facts were correctly stated, and I have been very careful to do so. I have also been very careful to state the results of the examination in a clear and concise manner.

I have been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner.

I have been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner.

I have been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner. I have also been very careful to state the results of the examination in a clear and concise manner.

Changing Format Within the Text

Input to PROSE:

```
.OPTION( K1 S2 ).MARGIN( K1 L10 R70 ).PARAGRAPH( K1 F& I2 S1 )
&There was nothing to be done. He had once heard a comedian
say,
.OPTION( K2 S1 ).MARGIN( K2 L20 R60 ).PARAGRAPH( K2 F& I0 S0 )
If you don't like the telephone company,
you know what you can do? Two tin cans and
a piece of string, that's what you can do.
That's the only alternative you've got.
.OPT 1.MAR 1.PAR 1.SKIP 1
The people in the audience had laughed. Eddie thought about
it now and decided it wasn't funny at all.
```

Output from PROSE:

There was nothing to be done. He had once heard a
comedian say.

If you don't like the telephone company,
you know what you can do? Two tin cans
and a piece of string, that's what you
can do. That's the only alternative
you've got.

The people in the audience had laughed. Eddie thought about
it now and decided it wasn't funny at all.

To change format for the next single-spaced, blocked paragraph, you would only need to enter:

```
.OPT 2.MAR 2.PAR 2
```

The example is a little more cumbersome than is necessary for one format change. Actually, you need only enter .OPT.MAR.PAR to return to keep buffer 1 in the text shown above. When no parameter is specified, the values are set to those stored in the numerically previous keep buffer, since the keep number is automatically incremented whenever a directive is entered and automatically decremented when that directive is entered without a parameter. Used in this way, the keep buffers function as a "stack" for temporary storage of variations from a basic format.

Reset Directive

The RESET directive sets twelve frequently changed directives to their default values. The following table summarizes the effect of the RESET directive on each:

1822

My dear Sir,
I have the honor to acknowledge the receipt of your letter of the 10th inst. in relation to the above mentioned subject. I am sorry to hear that you are not satisfied with the result of the examination. I have, however, no objection to your making such use of the facts as you may think proper.

I am, Sir, very respectfully,
Your obedient servant,
J. W. Alden

I am, Sir, very respectfully,
Your obedient servant,
J. W. Alden

I am, Sir, very respectfully,
Your obedient servant,
J. W. Alden

I am, Sir, very respectfully,
Your obedient servant,
J. W. Alden

I am, Sir, very respectfully,
Your obedient servant,
J. W. Alden

PROSE: A Text Formatter

<u>Directive</u>	<u>Effect</u>
INPUT	Default values for all options.
OPTION	Default values for all options.
FORM	Default values for all options, causes page eject.
COUNT	Sets page counter to 1.
TITLE	No titles until reentered.
SUBTITLE	No subtitles until reentered.
PAGE	Causes a page eject.
INDEX	Deletes all accumulated entries.
MARGIN	Default values for all options.
PARAGRAPH	No paragraphing until directive reentered.
OUTPUT	No pause at top of page; carriage return to do underlining; causes page eject.
SELECT	Discontinues page selection; all pages to be printed.

The **RESET** directive can be used three ways:

- Entering the directive name with no parameter resets the values of all directives to their defaults:
`.RESET`
- Using a directive name as a keyword resets the selected directive. For example, this command resets only the **MARGIN** and **OPTION** directives:
`.RESET( MARGIN OPTION )`
- Stating the keyword "except" and a directive name in the parameter excludes a selected directive. For example, this directive resets all directives with the exception of **FORM** and **OUTPUT**:
`.RESET( EXCEPT FORM OUTPUT )`

New directives may be entered after the **RESET** directive. The **RESET** directive is an easy way to clear a complicated series of format changes from the keep buffers.

Creating An Index

The **INDEX** and **SORTINDEX** directives provide the information PROSE needs to create an index for the document. The **INDEX** directive is entered as `.INX` to distinguish it from `.INDENT` and takes the remainder of the line together with the current page number as an index entry. As the formatted text migrates from page to page in various drafts, the page numbers in the index are updated.

Index entries accumulated by **INX** directives can be sorted alphabetically or by page number, then printed in a relatively flexible manner. The **SORTINDEX** directive allows you to specify the method of sorting entries and the format for printing the index.

The options for the **SORTINDEX** directive, listed in the following table, may be given in any order.

<u>Key Letter</u>	<u>Meaning</u>	<u>Default</u>
S	Sorting option. If this is numeric, it is the first significant column for alphabetic sorting. If it is the letter P, sorting is selected by page number.	1
M	Margin (left margin before index line)	0
P	Column (in index entry) to insert page number	0
L	Left width of page number (field width of number)	2
R	Right width of page number, blanks printed after	2

In the absence of a parameter, default values are used.

Printing the Document

This section does not anticipate all the possible idiosyncracies of the PDP-11 and its peripherals; it gives you directions for printing all or part of the document on certain standard devices.

Specifying Output Devices

The **OUTPUT** directive defines important aspects of the output device to which you will send the formatted text. The directive takes the general form:

.OUTPUT(device, options...)

All of the following is specific to the PDP-11.

One of the following acronyms indicates the output device to be used.

- ASC** ASCII terminals use the backspace for underlining, but are otherwise the same as the lineprinter (**LPT**) below. Pauses for page eject, however, are handled differently (see the **P** option below).
- LPT** Line printers use overprinting with a carriage return to do underlining. This is the default output device.

The following table contains the options for each output device. Keyword type is an integer or a switch.

<u>Key Letter</u>	<u>Meaning</u>	<u>Default</u>
E	Page eject at top of page ([in FORM description)	-
P	Pause at top of page	-
S	Shift output lines to the right	0
U	Underlining is available	+

These options may be given in any order.

- E:** The **page eject** option prints a form feed character for each time **PROSE** reads [in the **FORM** specification.
- P:** The **pause** option causes **PROSE** to stop printing and await operator acknowledgement each time a '[' character is encountered in the **FORM** specification. On an **ASC** terminal, **PROSE** will sound the bell and wait for a carriage return to be entered. For an **LPT** output device, no action will be taken.
- S:** The **shift** option shifts all **PROSE** output to the right by any number of spaces up to 50. This makes it easy to center output on a wide printer page.
- U:** If the destination terminal does not have underlining capability and the input file contains underline characters, the **underlining available** option should be turned off to prevent **PROSE** from trying to generate overprinted underlines.

Usually, the **OUTPUT** directive appears only at the beginning of the input file or in a header file. However, it must also be used immediately after a **.RESET(OUTPUT)** directive.

The **LITERAL** directive is useful for producing special printer control characters on some systems. It prints the remainder of the input line as a single output line, after special processing for upper and lower case, underlining, and literal blanks. This single line is printed independently of filling and justifying, or page-formatting processes; it is not counted as an output line.

PROSE: A Text Formatter

Printing Selected Pages and Sections

The **SELECT** directive will print specified pages of a document. Like the **OUTPUT** directive, the **SELECT** directive is placed before any lines that are to be printed on the output device, perhaps in a header file. Although the entire text will be formatted, only the selected pages will be printed, saving unnecessary printing time.

The parameter consists of page numbers separated by spaces. Two page numbers separated by a colon will select the span of pages, including beginning and ending numbers. As shown below, the plus sign '+' specifies a second page number relative to the first. The following example prints pages 3, 5, 10 through 15 inclusive, and 20 through 25 inclusive.

```
.SELECT( 3 5 10:15 20:+5 )
```

By default, all pages are printed.

Appendix A: Examples of PROSE Directives in Text

Example 1.

Input to PROSE:

```
.OUTPUT (LPT U+ E+)
.COMMENT +-----+
.COMMENT |NOTE: The header file supplied by Oregon Software |
.COMMENT |contains the output command for the line printer. |
.COMMENT |Do not use the above command if using that header. |
.COMMENT +-----+
.FORM ([ // #10 Pn #28 T /// L50 //// ]
+      [ // #60 Pn #17 S /// L50 //// ])
.COMMENT Page numbering starts at 62
.COUNT 62
.COMMENT This combination of form and count directives
.COMMENT duplicates the facing-pages format used in the
.COMMENT many typeset books.
.COMMENT
.TITLE The Programmer
.SUBTITLE Chapter Three
.MARGIN(L10 R62)
.INPUT (H/ U_)
.PARAGRAPH (F& I2)
&Computers, Eddie knew, have no idea where their sources of
information are in the world. They look upon the world as one great
big fat wire bulging with information and instructions, a wire with
no beginning, middle, or end. The world for a computer is merely an
electrical input saying, "Here's what you should know," or, "Here's
what I want to know," or, "Here's what you are to do now," and an
electrical output for them to talk back:
"Here's what I must know to answer your question," and, "Here are
your answers." To the computers, all interrogators and commanders
speak with the same voice and the same authority; all listeners have
the same ear.
&Like guns. It doesn't matter to a gun who pulls its trigger. Guns
have awesome power, but they are entirely dependent on the hands that
use them. Morally guns and computers are out of it all, though they
are regularly the instruments for people who make things happen.
&Twice now--first with Betty's parking ticket and now with his
$25,624.34--Eddie had been someone who made things happen. It was
a very exciting sen/sa/tion, one he hadn't previously experienced.
&He had seen, not long before on the "Today" show, an airline pilot
who talked about his af/fec/tion for the 747. He loved it more, he said,
than any other aircraft he had ever flown, and he had been a pilot for
twenty-five years. The inter/view/er asked him to explain his
enthusiasm.
&"I sit there in that little room in the front," the pilot said, "a
room just four stories off the ground when we're parked, but at the
top of the world when we're in flight--and I move little knobs and
dials. None of them takes more than a few ounces of pressure.
```

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

In an instant a machine weigh/ing a hundred tons responds more smoothly than if I were moving it myself. It's like the aircraft becomes an extension of myself because so little effort is needed to make it do what I want, and it does whatever I want it to do. It's very ex/cit/ing."

&"You make it sound almost sexual," the inter/view/er said.

&The pilot frowned. "I don't know about that. I never thought about that." His face brightened. "It's not sex. It's better. It's _real_ power."

&Eddie sensed that his entire relationship with the computer had started a radical change. Before, he had been the machine's servant, bringing it little orders and loads of information to feed upon. The questions weren't his and the answers never mattered to him. He was merely an intermediary in the affairs of others. Now he was having his own affair.

&And his own affair required further action before the check in his pocket became anything but a useless piece of paper.

.SKIP 2

&On his way back to the office Eddie stopped in the motor vehicle section. Edna was there alone, as usual, talking on the telephone, also as usual. She was wearing a different pink sweater. She held up her hand, the fingers all extended. At first Eddie thought she was showing off her rings--there were four of them, all different--but then he understood that she was telling him she would be on the phone five minutes longer. He waved his hand to indicate he was in no hurry. She leaned back in the chair, her pink breasts pointing toward the corner light fixture.

&Eddie wandered around the office, acting as if he were bored. He looked at some papers. She was paying no attention to him. He stopped at the drawer where blank drivers' licenses were kept. He looked over his shoulder. She was still talking, her back to him. Her left hand held the telephone and her right hand slowly rubbed the back of her neck.

&He quickly took from the drawer ten forms, then leaned on the counter and quietly stamped each of them with the tricolored state seal required for validation. He put the forms into his jacket pocket along with the two checks. Now he had only to type out whatever names he wanted to use and he would have official New York certification.

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that proper record-keeping is essential for the integrity of the financial system and for the ability to detect and prevent fraud. The document also outlines the responsibilities of those involved in the process, including the need for transparency and accountability.

The second part of the document provides a detailed overview of the various methods used to collect and analyze data. It describes the different types of data sources, such as surveys, interviews, and focus groups, and explains how this information is used to identify trends and patterns. The document also discusses the importance of ensuring the reliability and validity of the data collected.

The third part of the document focuses on the analysis and interpretation of the data. It describes the various statistical techniques used to analyze the data, such as regression analysis and correlation analysis, and explains how these results are used to draw conclusions. The document also discusses the importance of considering the context of the data and the potential limitations of the analysis.

Output from PROSE:

62

The Programmer

Computers, Eddie knew, have no idea where their sources of information are in the world. They look upon the world as one great big fat wire bulging with information and instructions, a wire with no beginning, middle, or end. The world for a computer is merely an electrical input saying, "Here's what you should know," or, "Here's what I want to know," or, "Here's what you are to do now," and an electrical output for them to talk back: "Here's what I must know to answer your question," and, "Here are your answers." To the computers, all interrogators and commanders speak with the same voice and the same authority; all listeners have the same ear.

Like guns. It doesn't matter to a gun who pulls its trigger. Guns have awesome power, but they are entirely dependent on the hands that use them. Morally guns and computers are out of it all, though they are regularly the instruments for people who make things happen.

Twice now--first with Betty's parking ticket and now with his \$25,624.34--Eddie had been someone who made things happen. It was a very exciting sensation, one he hadn't previously experienced.

He had seen, not long before on the "Today" show, an airline pilot who talked about his affection for the 747. He loved it more, he said, than any other aircraft he had ever flown, and he had been a pilot for twenty-five years. The interviewer asked him to explain his enthusiasm.

"I sit there in that little room in the front," the pilot said, "a room just four stories off the ground when we're parked, but at the top of the world when we're in flight--and I move little knobs and dials. None of them takes more than a few ounces of pressure. In an instant a machine weighing a hundred tons responds more smoothly than if I were moving it myself. It's like the aircraft becomes an extension of myself because so little effort is needed to make it do what I want, and it does whatever I want it to do. It's very exciting."

"You make it sound almost sexual," the interviewer said.

The pilot frowned. "I don't know about that. I never thought about that." His face brightened. "It's not sex. It's better. It's real power."

Eddie sensed that his entire relationship with the computer had started a radical change. Before, he had been the machine's servant, bringing it little

orders and loads of information to feed upon. The questions weren't his and the answers never mattered to him. He was merely an intermediary in the affairs of others. Now he was having his own affair.

And his own affair required further action before the check in his pocket became anything but a useless piece of paper.

On his way back to the office Eddie stopped in the motor vehicle section. Edna was there alone, as usual, talking on the telephone, also as usual. She was wearing a different pink sweater. She held up her hand, the fingers all extended. At first Eddie thought she was showing off her rings--there were four of them, all different--but then he understood that she was telling him she would be on the phone five minutes longer. He waved his hand to indicate he was in no hurry. She leaned back in the chair, her pink breasts pointing toward the corner light fixture.

Eddie wandered around the office, acting as if he were bored. He looked at some papers. She was paying no attention to him. He stopped at the drawer where blank drivers' licenses were kept. He looked over his shoulder. She was still talking, her back to him. Her left hand held the telephone and her right hand slowly rubbed the back of her neck.

He quickly took from the drawer ten forms, then leaned on the counter and quietly stamped each of them with the tricolored state seal required for validation. He put the forms into his jacket pocket along with the two checks. Now he had only to type out whatever names he wanted to use and he would have official New York certification.

Appendix A: Examples of PROSE Directives in Text

Example 2.

Input to PROSE:

```
.COMMENT Output directive is in the header file.
.INPUT( B# H/ )
.OPTION( K1 )
.FORM( [ // #50 W #80 E /// L50 / #30 '- ' PN:1 ' -' ] )
.MARGIN( K1 L5 R70 )
.PARAGRAPH( K1 F& S1 )
&Something clicked in another part of his mind and he knew he was about
to become a portable computerized superpower.
&The question had been puzzling him for some time. It had to do with
pro/gram access. If one had a pro/gram--if one knew the para/dig/matic
structure of a set of encoded information--then one could do nearly
any/thing one wanted with that information. If it was simply material
stored, then one could learn everything that was stored; if it was
operational material, then one could command the operations. The
problem was, one needed the program to do the work, and the utility
of the programs he had taken with him when he left Buffalo was
limited.
&The cold water swirled around his legs and the ripples moved out from
where his hands paddled the surface. Suddenly it was as if the answers
had typed themselves out on the console screen.
.OPT( K2 U+ S2 )
.MAR( K2 L15 )
.PARAGRAPH( K2 F& U8 S1 )
&QUESTION:##HOW DO I FIND OUT WHAT PROGRAMS EXIST
WHEN I DON'T KNOW WHAT QUESTIONS TO ASK?
&ANSWER:##ASK THE COMPUTERS WHAT QUESTIONS THEY CAN
ANSWER FOR YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.
.OPT.MAR.PAR
&He ran home through the woods without even bothering to
dry off. Mosquitoes pecked at his face. He dressed quickly,
hooked up the van, and sat down at his keyboard. He addressed
IFFI, the central law enforce/ment computer in Baltimore. The
acronym stood for Information#Filed#for#Future#Investigations.
He asked IFFI a question that translated as, "What discrete
sets of information have you on hand and what are the access codes
for them?" He set the machine for a printout rather than a
readout on the monitor.
&In seconds the Selectric began typing away. It typed for a long
time. Office Selectrics can handle about thirty characters a
second, faster than any human can go, but the ones built for
information processing went three times as fast. Nearly
one thousand five-/character units of information a minute, and
the machine seemed to be typing faster than he had ever seen it
go before...
```


PROSE: A Text Formatter

Output from PROSE:

3:35 PM 14 Jun 82

Something clicked in another part of his mind and he knew he was about to become a portable computerized superpower.

The question had been puzzling him for some time. It had to do with program access. If one had a program--if one knew the paradigmatic structure of a set of encoded information--then one could do nearly anything one wanted with that information. If it was simply material stored, then one could learn everything that was stored; if it was operational material, then one could command the operations. The problem was, one needed the program to do the work, and the utility of the programs he had taken with him when he left Buffalo was limited.

The cold water swirled around his legs and the ripples moved out from where his hands paddled the surface. Suddenly it was as if the answers had typed themselves out on the console screen.

QUESTION: HOW DO I FIND OUT WHAT PROGRAMS EXIST WHEN I DON'T
KNOW WHAT QUESTIONS TO ASK?

ANSWER: ASK THE COMPUTERS WHAT QUESTIONS THEY CAN ANSWER FOR
YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.

He ran home through the woods without even bothering to dry off. Mosquitoes pecked at his face. He dressed quickly, hooked up the van, and sat down at his keyboard. He addressed IFFI, the central law enforcement computer in Baltimore. The acronym stood for Information Filed for Future Investigations. He asked IFFI a question that translated as, "What discrete sets of information have you on hand and what are the access codes for them?" He set the machine for a printout rather than a readout on the monitor.

In seconds the Selectric began typing away. It typed for a long time. Office Selectrics can handle about thirty characters a second, faster than any human can go, but the ones built for information processing went three times as fast. Nearly one thousand five-character units of information a minute, and the machine seemed to be typing faster than he had ever seen it go before...

My dear Mr. [Name],

I have just received your letter of the 10th inst. and am glad to hear from you. I am well and hope this finds you the same. I have been thinking of you very much lately and wondering how you are getting on.

I have been very busy lately with my work, but I have managed to find some time to write to you.

I am sure you are doing well and hope to hear from you again soon.

Yours truly,

[Signature]

I am sure you are doing well and hope to hear from you again soon.

I am sure you are doing well and hope to hear from you again soon.

I am sure you are doing well and hope to hear from you again soon.

I am sure you are doing well and hope to hear from you again soon.

Appendix B: Summary Directive Table

Appendix B: Summary Directive Table

<u>Directive</u>	<u>Meaning (action)</u>	<u>Break</u>	<u>Parameter Type</u>	<u>Default</u>
BREAK	Break justification	*	-none-	-none-
COMMENT	No action		remainder of line	-none-
COUNT	Set page count		numeric	.COU 1
FORM	Define page format	*	(...)	.FOR([/2 T #62 E /3 L54 + /3 #33 '- ' PN:1 '- ' /4])
INDENT	Indent following line	*	numeric	-none-
INPUT	Specify input options	*	(...) or numeric	.INP(D.W150 K+1)
INX	Store index entry		remainder of line	-none-
LITERAL	Print literal text		remainder of line	-none-
MARGIN	Set margins	*	(...) or numeric	.MAR(LO R70)
OPTION	Set options	*	(...) or numeric	.OPT(S1 F+ M+ P+ L+ + R+ J3 E+ U- K+1)
OUTPUT	Specify output device		(...)	.OUT(LPT, SO U+)
PAGE	Eject to top of page	*	numeric	.PAG 5
PARAGRAPH	Set paragraphing params		(...) or numeric	-none-
RESET	Reset directive defaults	*	(...)	-none-
SELECT	Select pages to print	*	(...)	-none-
SKIP	Skip output lines	*	numeric	-none-
SORTINDEX	Sort and print index	*	(...)	.SOR(S1 L2 R2)
SUBTITLE	Set the subtitle		remainder of line	-none-
TITLE	Set the main title		remainder of line	-none-
UNDENT	Undent following line	*	numeric	-none-

The directives marked with an asterisk (*) cause a justification break before they are processed, since they affect the filling and justifying environment.

The ellipsis (...) indicates that the parameter is enclosed in parentheses and is described in detail along with the description of the directive itself.

Appendix C: Historical Notes

Most text-formatting programs available today descend from one of several original programs. Among these is RUNOFF, developed on the Dartmouth Time-Sharing System in the 1960s. Later, the Call-a-Computer system provided a RUNOFF version called EDIT RUNOFF as a text-editor command. In 1972, Michael Huck, working on the University of Minnesota's MERITSS system (a CDC 6400 running the KRONOS operating system), began to develop a version of EDIT RUNOFF called TYPESET.

TYPESET was intended as a "versatile text information processor commonly used to typeset theme papers, term papers, essays, letters, reports, external documentation . . . , and almost any other typewritten text."* In spite of these aspirations, no program can be all things to all people. TYPESET went through many changes, stabilizing somewhat in early 1977 at version 5.0, which is written in CDC COMPASS assembly language. John P. Strait developed PROSE, written in Pascal, over a year's time starting in the spring of 1977. The design was influenced heavily by TYPESET, making PROSE one of the many descendants of RUNOFF.

PROSE, with minor changes, was installed on the Univac 1100 series computers in early 1980 by Michael S. Ball of the Naval Ocean Systems Center. At Oregon Software, he converted this version from the Univac to the PDP-11 in July 1980 and to the Motorola MC68000 in the spring of 1982.

* Michael Huck, *Typeset 5.0 Information*, © 1977.

For More Information

We suggest several places to find more information about Pascal and the environment in which Oregon Software's products are used. Many of these books are available from Oregon Software. Prices are subject to change without notice.

Oh! Pascal by Doug Cooper and Mike Clancy.

An easy-to-read Pascal course for the novice programmer. (W. W. Norton, \$15.95)

Programming in Pascal by Peter Grogono.

A good course in standard Pascal, with lots of sample programs for experimentation. (Oregon Software supplies one copy to each new customer.)

Introduction to Pascal by Rodney Zaks.

A complete tutorial on Pascal designed to be read and understood by everyone. (SYBEX Inc., \$14.95)

A User Guide to the UNIX System by Rebecca Thomas and Jean Yates.

A beginner's tutorial on the UNIX operating system. (OSBORNE/McGraw-Hill, \$15.99)

UNIX Programmer's Manual Seventh Edition, Volume 2A.

A collection of tutorials on the use of the operating system and its utility programs; Brian Kernighan's descriptions of the system and the editor are especially helpful for new users. (Bell Laboratories, \$60.00 for Volumes 2A and 2B)

Pascal User Manual and Report by Kathleen Jensen and Niklaus Wirth.

The first definition of standard Pascal. (Oregon Software supplies one copy to each new customer.)

Algorithms + Data Structures = Programs by Niklaus Wirth.

A study of programming data structures, beginning with records, arrays, and sets — the fundamental structures — and progressing to those structures that are changed in value and structure by program execution. Full-length sample programs illustrate the stepwise refinements involved in developing Pascal programs. (Prentice-Hall, \$25.95)

Structured Programming by Dahl, Dijkstra, Hoare.

Three monographs on methodologies of concept modelling (Dijkstra), data structuring (Hoare), and structured, hierarchical programming (Dahl and Hoare). (Academic Press, \$20.00)

Elements of Programming Style by Kernighan and Plauger.

A practical demonstration of the principles of good programming and the use of common sense. The authors criticize and rewrite sample programs from various texts on programming. (McGraw-Hill, \$3.95)

Systematic Programming: An Introduction by Niklaus Wirth.

A description of the technique of constructing and formulating algorithms in a systematic manner, intended as general mathematical background rather than practical study of coding. (Prentice-Hall, \$23.95)

UNIX Programmer's Manual, Seventh Edition, Volumes 1 and 2B.

Volume 1 is the *Programmer's Reference*, containing explanations of all programs that make up the UNIX operating system. Volume 2B is the *Programmer's Manual*, a collection of papers on various aspects of installing and using the operating system for program development. Topics include various compilers, the assembler, and

For More Information

the UUCP network. (Bell Laboratories, \$100.00 includes Volumes 1, 2A, and 2B)

Concurrent Euclid, the UNIX System, and Tunis by Holt, Graham, Lazowska, Scott.

An introductory text on concurrent programming, the techniques used to design and implement operating systems, computer networks, real-time control and embedded microprocessor systems. (Addison-Wesley, \$15.95)

Pascal Newsletter

Published quarterly, Oregon Software's *Pascal Newsletter*, which contains status reports on all of our Pascal products, announcements of new versions of software and new products, and various technical articles.

Temporarily, the newsletter is the sole publication for the Oregon Pascal Users Society (OPUS), an organization dedicated to the sharing of information between Oregon Software and its customers. Oregon Software is not affiliated with OPUS, but we encourage its activities and provide space for an OPUS column in our *Pascal Newsletter*, until OPUS begins to publish its own newsletter. OPUS membership is free. To join, write:

Oregon Pascal Users Society (OPUS)
Bruce Williams
c/o EOCOM
15771 Redhill Ave.
Tustin, California 92680
(714) 730-5051, ext. 302.

The Pascal Newsletter

Published by the Pascal Users' Group, *The Pascal Newsletter*, is available at \$10 for a one-year subscription. Contact:

Pascal Users' Group
2903 Huntington Rd
Cleveland, Ohio 44120.

Index

Style note: Page numbers in boldface (b) indicate the defining use of the term.

A

alignment, byte boundaries, 2-33
allocating memory, 2-24
alphabetize, *see* structure, manual
ASCII format, 3-6
assembly language, *see* PASMAL

B

bitsize function, 2-34, **3-21**
Blaise Pascal, 1-2
boolean operators, 3-18
boundary, alignment of user-defined types,
2-33
breakpoint, 1-7, *see also* Debugger, breakpoint
commands
break procedure, 3-16
/buff:n I/O control switch, **2-9**
Bug Reports, *see* Trouble Reports

C

cache memory, 2-28
case statement, 3-13
 example with otherwise, 3-13
 otherwise clause, 3-13
chaining, 2-23
channels, input and output, 2-19
 overlay load, 2-16
characters, data type, 3-6
character strings, *see* Dynamic String Package
 literal, 3-4
check compilation switch, 2-3
\$check embedded switch, 2-7
chr function, 3-30
Clancy, Mike, Info-1
close procedure, **3-15**, 3-26
colon notation, 2-2, 2-5
command lines, reading of, 2-46
comment PB switch, 5-9
compilation, 1-1
 errors, 1-2, 2-62, *see also* error mes-
 sages
 examples, 1-1, 2-7
 syntax, 2-1
compilation switches, 1-4, **2-2**
 check switch, 2-3

debug switch, 1-6, 2-3, 2-60, 4-2
double switch, 1-8, 2-2, 2-18
errors switch, 1-4, 2-3
list switch, 1-4, 2-3
macro switch, 2-3, 2-19
main switch, 2-2
no- reverses effect of <switch>, *see*
 <switch>
object switch, 2-3
own switch, 2-2, 3-8
profile switch, 1-8, 2-3, 4-32
standard switch, 2-3
test switch, 2-3
times switch, 2-3
walkback switch, 2-2
workspace switch, 2-2
pascal1 switch, 2-2
compiler directives, \$include, 3-8
compiler errors, 2-76
 consistency checks, 2-76
 overflow, 2-76
compiler installation, *see* Release Notes
compiler optimizations, *see* optimization
concatenation of source files, 2-1, 2-23,
5-18
conformant array parameters, xiv, 3-2
 examples, 3-4
 syntax, 3-3
constants, structured, 3-9
CONSTS psect, 2-22
Cooper, Doug, 1-1, Info-1
cross-reference utilities, 5-13
 PROCREF procedure lister, 5-15
 XREF identifier lister, 5-13
customized error messages, 2-39

D

Dahl, O.-J., Info-1
debug compilation switch, 1-6, 2-3, 2-60,
4-2
\$debug embedded switch, 2-5
Debugger, 4-1
 assigning values to variables, 4-14
 breakpoint commands, 4-6
 breakpoints, 1-7
 command summary, 4-31
 data commands, 4-13
 debug compilation switch, 4-2

Index

- debugging external modules, 4-26
- example, 1-6
- execution control commands, 4-8
- execution history, 4-11, 4-20
- information commands, 4-17
- initialization of, 2-19
- overlaying of, 4-30
- stack commands, 4-20
- summary of commands, 4-31
- tracking commands, 4-11
- utility commands, 4-18
- declaration sections, interleaving of, 3-8
- delete** procedure, **3-26**
 - example, 3-26
- deleting files, *see delete* procedure
- DIAGS psect, 2-22
- Dijkstra, E.W., Info-1
- direct access, 2-10, 3-15
- dispose** procedure, 2-24, 2-28, 3-30
- disposing of memory, 2-24
- double** compilation switch, 2-2, 2-18
 - example, 1-8
- \$double** embedded switch, 2-5, 2-18
- double precision, 2-2, 2-5, 2-18
 - colon notation, 2-2, 2-5
 - example, 1-8
- dynamic link, 2-26
- Dynamic String Package, 2-18, 5-18
 - and **%include** directive, 5-18
 - capabilities, 5-18
 - example, 5-20
 - library creation, 2-18
 - procedures and functions, 5-19
 - STRING, 5-18
 - string declaration, 5-18
 - string examples, 5-18
 - use of, 5-20

E

- EBNF syntax diagrams, 3-40
 - notation, 3-38
- eis** processor switch, 2-4
- embedded switches (**\$**), **2-4**
 - \$check** switch, 2-7
 - \$debug** switch, 2-5
 - \$double** switch, 2-5, 2-18
 - examples, 2-4
 - \$indexcheck** switch, 2-7
 - \$list** switch, 2-6
 - \$main** switch, 2-5
 - \$no-** reverses effect of **<switch>**,
see <switch>
 - \$own** switch, 2-5
 - \$pascal1** switch, 2-5
 - \$pointercheck** switch, 2-7
 - \$profile** switch, 2-6
 - \$rangecheck** switch, 2-7
 - \$stackcheck** switch, 2-7
 - \$standard** switch, 2-6
 - \$walkback** switch, 2-5
- entry points, list of, 2-78
- p\$gtla**, 2-47
- support library, 2-19
- error correction, *see* Debugger
- error handling, 2-39
- error messages, 1-2, 1-3
 - compilation, 1-2, 2-61
 - compiler, 2-76
 - customized, 2-39
 - run-time, 1-3, 2-35, 2-72
- error recovery, run-time, 3-14
- errors, 1-2
 - compilation, 1-2
 - compiler, 2-76
 - fatal, 2-35
 - I/O, 2-35
 - run-time, 1-3, 2-35, 3-14
 - run-time detection, 2-39
- errors** compilation switch, 1-4, 2-1, 2-3
- error trapping, I/O errors, 2-37
- error walkback, 2-2, 2-5
- executable file, 1-1
- execution history, *see* Debugger
- exitst** procedure, 2-43
 - declaration, 2-43
 - status values, 2-43
- extended-instruction set, 2-4
- extended-range integers, *see* unsigned integers
- extended-range variables, 3-18
- Extended Backus-Naur Form, *see* EBNF
 - syntax diagrams
- extended precision, 2-2, 2-18
- extensions, default, 2-77
 - I/O support, 3-14
 - language, 3-8
 - syntax, 3-8
- external** directive, 2-11, 3-9
- external file access, 3-14
- external libraries, **2-14.4**
 - header file, 2-14.4
- external modules, **2-11**, 3-8, 3-9
 - and double precision, 2-18
 - debugging of, 4-26
 - definition, 2-11
 - definition example, 2-11
 - example program, 2-12
 - FORTTRAN routines, 2-11, 2-14, 2-14.2

- global variable reference, 2-11
- identifiers, 2-11
- linking errors, 2-12
- MACRO routines, 2-11, 2-14
- `$nomain` embedded option, *see* embedded switches
- `nomain` option, *see* compilation switches
- external procedures, 5-23

F

- file, 1-1
 - closing upon procedure exit, 2-44
 - cross-reference, 5-13
 - default extensions, 2-77
 - executable, 1-1
 - "header", 5-18
 - input, 2-1
 - listing, 1-4, 2-1
 - object, 1-1
 - output, 2-1
 - overlay description, 4-30
 - profile, 1-8, 4-32
 - PROSE, 5-42
 - source, 1-1
 - statement map, 1-6
 - symbol, 1-6
 - temporary, 2-10
 - text, 3-6, 3-30
- file buffer, 2-9
 - default size, 2-9
 - size of, 2-9
- file control switches, *see* I/O control switches
- file dump, example, 2-41
- file extensions, default, 2-77
- files, renaming of, 3-26
- `fis` processor switch, 2-4
- floating-point numbers, format of, 3-6
 - outputting of, 3-17
- foreground operation, 2-49
- formatter, *see* PASMAT
- for statement, 3-30
- FORTTRAN, called from Pascal, 2-11, 2-14, 2-14.2
 - restrictions, 2-14.3
- forward directive, 2-12
- `fpp` processor switch, 2-3
- `FRUN` command, 2-49
- function return value, 2-26
 - structured types, 3-25

G

- `getlin` procedure, 2-46
 - declaration, 2-47
 - example, 2-49
 - parameters, 2-47
- `getpos` procedure, 2-54, 3-16
 - example, 2-55
 - parameters, 2-54
- `get` procedure, 2-10, 3-15, 3-30, 3-31
- GLOBAL psect, 2-22
- `/go` I/O control switch, 2-9
- Grogono, Peter, 1-1, Info-1
- "grow" psects, 2-17

H

- heap, 2-24, 2-27
 - fragmentation of, 2-28
 - free list, 2-17
 - `P$KORE`, 2-28
- Hoare, C. A. R., Info-1

I

- I/O control switches, 2-9, 3-14
 - `/buff:n` switch, 2-9
 - examples, 2-9
 - `/go` switch, 2-9
 - `/nfs` switch, 2-9
 - `/odt` switch, 2-9
 - `/seek` switch, 2-10
 - `/size:n` switch, 2-10
 - `/span` switch, 2-10
 - switch summary, 2-9
 - `/temp` switch, 2-10
- I/O errors, 2-9
 - trapping of, 2-9
- I/O error trapping, 2-37
- I/O functions, 2-50
 - and standard Pascal, 2-50
 - lazy I/O, 2-50
- I/O page, 2-25
- I/O support extensions, 3-14
- identifier, predefined, 3-32
- identifiers, extension to standard, 3-8
 - `maxint`, 3-6
 - valid characters, 3-5
- `%include` directive, 2-19, 2-40, 2-44, 3-8, 5-18
 - examples, 2-44
 - syntax, 2-44, 3-8
- `indent` PB switch, 5-9
- index, *see* alphabetize
- `$indexcheck` embedded switch, 2-7, 3-30

Index

initializing the support library, 2-27
initializing variables, 2-46
input file, 2-1
installation, compiler, *see* Release Notes
integer overflow, 3-31
integers, range of values, 3-18
 unsigned, 2-57, 3-6, 3-18
interrupt vectors, 2-23
ioerror function, 2-37, 2-55
iostatus function, 2-37, 2-38.2, 2-55
ISO Standard Pascal, xiv, 3-1
 alternate symbols, 3-5
 implementation definitions, 3-6
 Pascal-2 extensions, 3-8
 recent changes, 3-1

J K

Jensen, Kathleen, 1-1, 3-1, Info-1
Kernighan, Brian, Info-1

L

language extensions, 3-8
 structured constants, 3-9
lazy I/O, 2-49
Librarian, 2-18
 examples, 2-18
libraries, run-time, 2-23
 support, 2-19
 system macro, 5-36
LINK command, 2-15
 example, 2-8
Linker, 1-1, 2-15, *see also* overlays
list compilation switch, 2-3
 example, 1-4
\$list embedded switch, 2-6
listing, 1-4
 file, 1-4
 page heading, 1-4
listing file, 2-1
literal strings, 3-4
LOAD command, 2-25
look-ahead, 2-27
loophole function, 3-21
 example, 3-22

M

MAC command, 5-37
MACRO-11, *see* PASMAL
 called from Pascal, 2-14
MACRO assembler command, 2-3, 5-37
macro compilation switch, 2-3, 2-19
macro package, *see* PASMAL

main compilation switch, 2-2
\$main embedded switch, 2-5
manual, index, *see* index
manual purpose, xi
maxint identifier, 3-6, 3-30
memory, typical arrangement, 2-27
 typical layout, 2-23
memory allocation, 2-24
 user-defined types, 2-33
memory map, 2-40
 example, 2-41
memory usage, monitoring of, 2-27
/M:n Linker option, 2-27
mod function, 3-25, 3-30
monitoring memory usage, 2-27
multiple input files, 2-1
multiple source files, 2-44

N

NewOK function, 2-27, 2-30
new procedure, 2-24, 2-27, 2-28, 3-30
/nfs I/O control switch, 2-9
nil pointer, 2-7
no- reverses effect of <switch>, *see* <switch>
noioerror function, and /go switch, 2-9
noioerror procedure, 2-37
non-standard features, 3-25
 program parameters, 3-25
 returning of structured types, 3-25
nondecimal notation, 3-18
nonpascal directive, 2-11, 2-14, 2-52,
 3-9
null root segments, 2-17

O

object compilation switch, 2-3
object file, 1-1
object module libraries, *see* Librarian
Octal Debugging Technique (ODT), 2-9
octal notation, 3-18
octal output, 3-16
/odt I/O control switch, 2-9
 examples, 2-10
OPERRO.PAS, 2-39
optimization, 2-59
 base address calculation, 2-60
 Boolean evaluations, 2-59
 constant folding, 2-59
 dead code elimination, 2-59
 expression targeting, 2-60
 redundant branching, 2-60
 redundant expressions, 2-60
 register assignments, 2-59

1890

Jan 1st to Dec 31st

1891

1892

1893

1894

1895

1896

1897

1898

1899

1900

1901

1902

1903

1904

- options** PASMAT switch, 5-3
- Oregon Pascal Users Society (OPUS), Info-1
- Oregon Software, xl, 2-76
 - Pascal Newsletter, Info-2
 - Trouble Reports, 2-76
- organization, *see* manual, index
- origin** declaration, 3-20
 - examples, 3-20
 - syntax, 3-20
- otherwise** clause, 3-13
- output specifications, 2-1
 - colon notation, 2-2, 2-5, 2-18
 - default field widths, 3-7
 - octal notation, 3-16, 3-18
 - scientific notation, 3-7, 3-17
- overlays, 2-16
 - and **getlin**, 2-47
 - Debugger, 4-30
 - examples, 2-16
 - extended memory, 2-16, 2-17
 - load channel, 2-16
 - low memory, 2-16
 - null root segments, 2-17
 - /O:n Linker switch, 2-16
 - program segments, 2-16
 - virtual, 2-16, 2-17, 2-27
 - virtual job, 2-15
- own** compilation switch, 2-2, 3-8
 - and GLOBAL psect, 2-22
- \$own** embedded switch, 2-5

P

- P\$AREA** temporary storage, 2-47
- %page** directive, syntax, 3-9
- page** procedure, 3-7
- parameters
 - conformant array parameters, 3-2
 - passing of, 2-26
 - procedure and function, 3-2
- Pascal, Blaise, 1-2
- pascal1** compilation switch, 2-2
- \$pascal1** embedded switch, 2-5
- Pascal standard, xiv, *see also* ISO Standard Pascal
- Pascal Users' Group, Info-2
- PASMAT, 2-14.1, 3-29, 5-23
 - and system macro library, 5-36
 - begin** macro, 5-29
 - command line, 5-25
 - design of MACRO-11 procedures, 5-23
 - endpr** macro, 5-30
 - examples, 5-24, 5-33
 - func** macro, 5-27
 - macro definitions, 5-24

- param** macro, 5-27
- placing in system macro library, 5-36
- predefined types, 5-31
- proc** macro, 5-26
- purpose of, 5-23
- rsave** macro, 5-29
- save** macro, 5-28
- var** macro, 5-28
- PASMAT, 5-2
 - command line, 5-3
 - directives, 5-4
 - examples, 1-4, 5-7
 - limitations, 5-6
 - options** switch, 5-3
 - portability mode, 5-5
- PB formatter, 5-9
 - comment** switch, 5-9
 - example, 5-10
 - formatting rules, 5-12
 - indent** switch, 5-9
- P\$CODE** psect, 2-22
- p\$dispose** procedure, 2-27, 2-28
 - declaration of, 2-29
 - example, 2-30
 - parameters, 2-29
- P\$DYNL** psect, 2-22
- P\$GROW** psect, 2-17
- P\$GRWH** psect, 2-17
- p\$gtln** entry point, 2-47, *see also* **getlin** procedure
- p\$inew** function, 2-27, 2-28
 - declaration of, 2-29
 - example, 2-30
 - parameters, 2-29
- P\$KORE** psect, 2-17
- PMA, 2-22, 2-40
- \$pointercheck** embedded switch, 2-7, 3-30
- pointers, 2-7
 - nil**, 3-30
 - nil** values, 2-7
 - stack pointer, 5-27
 - use of, 2-56
- Post-Mortem Analyzer (PMA), 2-22, 2-40
- predefined functions and procedures, 3-26
- predefined identifiers, 3-32
- pred** function, 3-30
- procedure walkback, 1-3, 2-2, 2-5, 2-35, *see also* walkback
- processor switches, 2-3
 - ois**, 2-4
 - fis**, 2-4
 - fpp**, 2-3
 - sim**, 2-4

1. The first part of the report
describes the general situation
of the country and the
state of the economy.
It also mentions the
main problems which
the government is
facing at the present
time.
2. The second part of the
report deals with the
financial situation of the
country. It gives a
detailed account of the
revenue and the
expenditure of the
government for the
last year.
3. The third part of the
report discusses the
social and cultural
conditions of the
country. It mentions
the progress made in
the field of education,
health and social
welfare.
4. The fourth part of the
report deals with the
foreign relations of the
country. It mentions
the main events which
have taken place in
the field of international
relations during the
last year.
5. The fifth part of the
report discusses the
prospects for the
future. It mentions
the main problems which
the government is
facing and the
steps which are being
taken to solve them.

6. The sixth part of the
report deals with the
internal security of the
country. It mentions
the main events which
have taken place in
the field of internal
security during the
last year.
7. The seventh part of the
report discusses the
prospects for the
future. It mentions
the main problems which
the government is
facing and the
steps which are being
taken to solve them.
8. The eighth part of the
report deals with the
internal security of the
country. It mentions
the main events which
have taken place in
the field of internal
security during the
last year.
9. The ninth part of the
report discusses the
prospects for the
future. It mentions
the main problems which
the government is
facing and the
steps which are being
taken to solve them.
10. The tenth part of the
report deals with the
internal security of the
country. It mentions
the main events which
have taken place in
the field of internal
security during the
last year.

Index

PROCREF utility, 5-15
 command line, 5-15
 example, 5-16
 limitations, 5-16
profile compilation switch, 2-3, 4-32
 example, 1-8
\$profile embedded switch, 2-6
profile file, 1-8
Profiler, 4-32
 example, 1-8
 execution requirements, 4-32
 output explanation, 4-33
 usage of, 4-32
 warnings, 4-36
program heading, 3-8
 file parameters, 3-8
program sections ("psects"), 2-22
 attributes, 2-22
 CONSTS psect, 2-22
 DIAGS psect, 2-22
 GLOBAL psect, 2-22
 P\$CODE psect, 2-22
 P\$DYNL psect, 2-22
 P\$GROW psect, 2-17
 P\$GRWH psect, 2-17
 P\$KORE psect, 2-17
 SHIFTS psect, 2-22
 TABLES psect, 2-22
program segments, and overlays, 2-16
PROSE, 5-20, 5-38
 basic text units, 5-39
 command line, 5-42
 directive format, 5-40
 directive summary, 5-63
 examples, 5-41, 5-52, 5-57
 index directives, 5-54
 input control directives, 5-43
 "keep" buffer, 5-52
 miscellaneous options, 5-44
 output control directives, 5-55
 page control directives, 5-47, 5-48
 paragraphs, 5-49
 parameter values, 5-40
 title directives, 5-47
psects, *see* program sections
purpose of manual, xi
put procedure, 2-10, 3-15, 3-31

R

random access, 3-15, 2-10, *see also* **getpos**
 and **setpos** procedures
 simulated, 2-54, 3-16
\$rangecheck embedded switch, 2-7, 3-30
reading command lines, 2-46

readln procedure, 3-16, 3-30
read procedure, 3-16, 3-30
real numbers, 2-18
 format of, 3-6
 outputting of, 3-17
recursion, 2-36
ref function, 3-21
registers, 2-26
 saving of, 5-28
rename procedure, 3-26
 example, 3-26
renaming files, *see* **rename** procedure
reserved words, 3-32
reset procedure, 2-9, 2-38, 3-7, 3-14,
 3-26, 3-30
 parameters, 3-14
 syntax, 3-14
return link, 2-26
rewrite procedure, 2-9, 2-38, 3-7, 3-14
 and **size:n** I/O control switch, 2-10
 parameters, 3-14
 syntax, 3-14
round function, 2-58
R PASCAL command, 2-1
run-time error messages, 2-72
run-time error reporting, customizing of,
 2-39
 OPERRO.PAS, 2-39
 UERROR.PAS, 2-39
run-time errors, 1-3, 2-35
 procedure walkback, 1-3
 recovery from, 3-14
 walkback, 1-3, 2-35
run-time library, 2-23

S

sayerr procedure, 2-37, 2-38.1, 2-40
 example, 2-38.2
scientific notation, 3-7, 3-17
/seek I/O control switch, 2-10, 3-15,
 3-16, 3-31
 example, 3-16
seek on text files, examples, 2-55
 simulated, 2-54
seek procedure, 2-10, 2-54, 3-15
segments, and overlays, 2-16
setpos procedure, 2-54, 2-55, 3-16
 example, 2-55
 parameters, 2-55
sets, 3-6, 3-31
 use of, 5-20
 .SETTOP system directive, 2-27
SHIFTS psect, 2-22
sim processor switch, 2-4, 2-22

- single-character mode, 2-9
- single precision, 2-5, 2-18
- size function, 2-30, 2-34, **3-21**
- /size:n I/O control switch, **2-10**
- source, 1-1
 - file, 1-1
 - program, 1-1
- source file concatenation, **2-23**, 5-18
- source program, 1-1
- space function, 2-27, **2-28**
 - data types, 2-28
 - declaration of, 2-28
 - example, 2-30
- /span I/O control switch, **2-10**
- stack, 2-19, 2-27, 5-23
 - contents, 2-24
 - default size, 2-27
 - reserving space, 2-30
 - space function, 2-28
 - stack frame, 2-25
- \$stackcheck embedded switch, 2-7
- stack frame, **2-25**
- stack pointer, 2-17, 2-24, 5-27
- standard, *see* ISO Standard Pascal
- standard compilation switch, 2-3
- \$standard embedded switch, 2-6
- standard Pascal, xiv, *see also* ISO Standard Pascal
- START.OBJ, and virtual overlays, 2-17
- statement map file, 1-6
- storage allocation, 2-33
 - pre-defined types, 2-33
 - size and bitsize, 2-34
 - user-defined types, 2-33
- string processing, 5-18
- strings, *see* Dynamic String Package
 - declaration of, 3-16, 5-18
 - examples, 5-18
 - literal, 3-4
- structure, manual, *see* organization
- structured constants, 3-9
 - examples, 3-10
 - multidimensional arrays, 3-12
- style notes, xvi
- succ function, 3-30
- support library, 2-19
 - data definitions, 2-19
 - entry points, 2-19, 2-78
 - error-control module, 2-39
 - initialization procedure, 2-27
 - initializing of, 2-19
 - LIBDEF.PAS, 2-19
 - library work area, 2-19
- switches, 2-2, 2-4, 2-18, *see also* I/O con-

- trol switches, compilation switches, embedded switches, processor switches
- compilation, 2-2
- compilation examples, 2-7
- embedded (\$), 2-4
- I/O control, 2-9
- processor, 2-3
- symbol file, 1-6
- syntax diagrams, Pascal-2, 3-33
- system device (SY:), 2-15, 2-46
- system macro library, and PASMAL, 5-36

T

- TABLES psect, 2-22
- /temp I/O control switch, **2-10**
- termination status, *see* exitst procedure
- test compilation switch, 2-3
- text files, 3-6
- text formatting, *see* PROSE
- Thomas, Rebecca, Info-1
- time function, **3-27**
 - example, 3-27
- times compilation switch, 2-3, 2-46
- timestamp procedure, 3-28
 - example, 3-28
 - parameters, 3-28
- traceback, *see* walkback, procedure
- Trouble Reports, 2-76
- trunc function, 2-58
- two's complement arithmetic, 2-57
- type checking, suppression (loophole), 3-21
- type coercion, *see* loophole function

U

- UERROR.PAS, 2-39
- Ufloat procedure, unsigned floating-point conversion, 2-57
- unsigned integers, 2-28, **2-57**, 3-6, 3-18
 - floating-point conversion, 2-57
 - outputting of, 2-57
 - subrange notation, 2-57
- user service routine (USR), 2-25
- utility package, **5-1**
 - Dynamic String Package, 5-18
 - PASMAL, 2-14.1, 3-9, **5-23**
 - PASMAT, 3-9, 5-2
 - PB formatter, 5-9
 - PROCREF, 5-15
 - PROSE, 5-38
 - XREF, 5-13
- Utrunc function, unsigned truncation, 2-58
- Uwrite procedure, unsigned write, 2-57

[Faint, illegible handwritten text, likely bleed-through from the reverse side of the page.]

[Faint, illegible handwritten text, likely bleed-through from the reverse side of the page.]

Index

V

variable initialization, 2-46
VIRJOB.OBJ, 2-15
virtual job, example, 2-17
 START.OBJ, 2-17
 VIRJOB.OBJ, 2-15
virtual jobs, 2-15
 job status word (JSW), 2-15
virtual overlays, 2-27, *see also* overlays, extended
 memory

W

walkback, xiv, 2-35, 3-15
 disabling of, 2-35
 examples, 2-36
 procedure, 1-3, 2-2, 2-35
 psect, 2-22
 run-time, 2-35
walkback compilation switch, 2-2
\$walkback embedded switch, 2-5
Wirth, Niklaus, 3-38, Info-1
WK: logical device, 2-46
work files, example assignment, 2-46
 location of (WK:), 2-46
workspace compilation switch, 2-2
writels procedure, 3-4
write procedure, 3-4
write statement, double-precision values,
 2-2

X Y Z

XM monitor, virtual jobs, 2-15
XREF utility, 5-13
 command line, 5-13
 example, 5-14
Yates, Jean, Info-1
Zaks, Rodney, Info-1

Documentation Evaluation Report

Pascal-3 V2.1 User Manual Update Package No. 2
PDP-11/RT-11

Good documentation is as important as good software. We at Oregon Software are well aware that you expect both, so we value your responses. Use this form to write down comments and suggestions, which will help us improve the quality and usefulness of our publications. If you require a written response, submit your comments on a Trouble Report.

Rate the following items on a scale of 1 to 5, with 5 being the highest rating.

☐ Initial impression	☐ Ease of finding information
☐ Organization	☐ Accuracy
☐ Ease of reading	☐ Ease of updating the manual

What changes in the documentation, in your opinion, are most useful? _____

What aspects of the update package need improvement? _____

What errors have you found in this update package? Include page numbers. _____

Name _____ Site # _____
Company _____ Date _____

Journal of the [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Mailing Instructions

Domestic. After filling out the report, cut along the vertical dotted line to remove the punched holes. Then fold this page so these instructions are hidden and the BUSINESS REPLY MAIL permit faces out. Secure the loose fold with staple or tape, then mail.

International. Please enclose this page in an envelope and return it to your distributor. If you do not have a distributor, return the report directly to us. (Unfortunately, we cannot supply prepaid postage for overseas respondents.)

Thank you in advance for your response.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

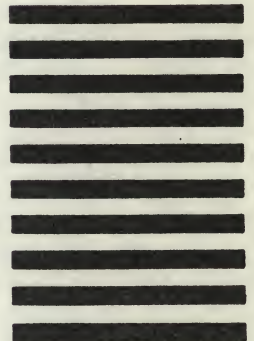
FIRST CLASS

PERMIT NO. A407

PORTLAND, OR.

POSTAGE WILL BE PAID BY ADDRESSEE

OREGON SOFTWARE, INC.
Attn: Documentation Comments
6915 SW Macadam
Portland, OR 97219



Handwritten title or header at the top of the page.

First paragraph of handwritten text, appearing as a list or series of entries.



Vertical column of handwritten text on the left side of the page.

Second main paragraph of handwritten text in the center of the page.

Large section of handwritten text at the bottom of the page, possibly a concluding paragraph or a detailed list.

Release Package Checklist

SOFTWARE: Pascal-2 Development System

VERSION: V3.1D

OPERATING SYSTEM: RT-11

DATE OF RELEASE: January 15, 1985

This release package contains the following marked items. If a discrepancy exists between the marked items and the contents of your release package, please contact Oregon Software at (503) 245-2202 immediately.

- ☒ Pascal-2 software on:
 - ☐ Magnetic tape. ☐ 800 bpi ☐ 1600 bpi
 - ☒ Floppy disk.
 - ☐ Cartridge disk.
- ☒ *Pascal-2 User Manual*, Second Edition.
- ☒ Documentation Update Package, Update No. 2.
- ☒ Release Notes, including the Installation Guide.
- ☒ *Standard Pascal User Reference Manual* by Doug Cooper.
- ☒ *Programming in Pascal* by Peter Grogono.
- ☒ An assortment of *Pascal Newsletters*.
- ☒ Extra Trouble Report forms.
- ☐ Extra "Request to Amend" forms.
- ☐ License agreement.

THE FIRST PART OF THE HISTORY OF THE REIGN OF HENRY THE SEVENTH

By Sir Thomas More
KING OF ENGLAND
COUNSELLOR
AND CHANCELLOR

With a new edition of the life of Sir Thomas More
by Sir Thomas More

By Sir Thomas More
KING OF ENGLAND
COUNSELLOR
AND CHANCELLOR

With a new edition of the life of Sir Thomas More
by Sir Thomas More

By Sir Thomas More
KING OF ENGLAND
COUNSELLOR
AND CHANCELLOR

Release Notes

Pascal-2 V2.1D for RT-11

January 15, 1985

The software described by this publication is subject to change without notice. Oregon Software assumes no responsibility for the use or reliability of any of its software that is modified without the prior written consent of Oregon Software.

Oregon Software holds right, title, and interest in the software described herein. The software, or any copies thereof, may not be made available to or distributed to any person or installation without the written approval of Oregon Software.

This publication, or parts of it, may be copied for use with the licensed software described herein, provided that all copies include this notice and all copyright notices.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-division (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013 of the Federal Acquisitions Regulations (FARs).

Name of Contractor and Address:

Oregon Software, Inc.
6915 S.W. Macadam Avenue
Portland, Oregon 97219
Phone: 503-245-2202

© 1985 Oregon Software, Inc. All Rights Reserved. Printed in USA, January 1985.

Pascal-1, Pascal-2 and Oregon Software are trademarks of Oregon Software, Inc.

DEC, PDP, RSX, RSTS/E and RT-11 are trademarks of Digital Equipment Corporation.

THE UNIVERSITY OF CHICAGO
LIBRARY

1950

THE UNIVERSITY OF CHICAGO LIBRARY
1207 EAST 58TH STREET, CHICAGO, ILL. 60637

ALL BOOKS ARE LOANED FREE OF CHARGE
TO THE MEMBERS OF THE LIBRARY

FOR THE USE OF THE LIBRARY
AND THE MEMBERS OF THE LIBRARY

THE UNIVERSITY OF CHICAGO LIBRARY
1207 EAST 58TH STREET, CHICAGO, ILL. 60637

ALL BOOKS ARE LOANED FREE OF CHARGE
TO THE MEMBERS OF THE LIBRARY

FOR THE USE OF THE LIBRARY
AND THE MEMBERS OF THE LIBRARY

THE UNIVERSITY OF CHICAGO LIBRARY
1207 EAST 58TH STREET, CHICAGO, ILL. 60637

ALL BOOKS ARE LOANED FREE OF CHARGE
TO THE MEMBERS OF THE LIBRARY

FOR THE USE OF THE LIBRARY
AND THE MEMBERS OF THE LIBRARY

Pascal-2 V2.1D/RT-11 Release Notes

The information contained in this document describes the Pascal-2 V2.1D release package. In these notes you will find:

- A list of problems ("bugs") we've fixed in this release.
- Notes on changes in the documentation and on how to update the user manual.
- Miscellaneous notes of interest to Pascal-2 users.
- The Installation Guide.

These release notes should be read before attempting to install the compiler and utilities on your system. The "Documentation Notes" section of the release notes should be read before attempting to insert the change pages from the update package into the user manual.

Style Notes

This document follows these style conventions:

Text:

Pascal reserved words, predefined symbols, switches and compiler directives are in boldface typewriter: **begin**, **write**, **include**, **main**. Portions of examples referred to in the text appear in boldface typewriter (the type style used in examples). Program, system and file names are in upper-case letters in the same type style as the text: **SAMPL**, **VMS**, **SAMPL.PAS**.

Program Examples:

Commands that you should enter are in underlined boldface typewriter: **RUN EX**. These commands assume a carriage return at the end.

Program Listings:

The Pascal-2 compiler accepts any combination of upper-case and lower-case characters. Examples in this manual have Pascal words in lower case and have user-defined words with an initial capital letter and other capitalization as needed for readability, as shown in this program fragment:

```
procedure Show;  
begin  
    SomeUserAction;  
    writeln(Result);  
end;
```

Terminology:

We use standard terms as they are used in documents describing the RT-11 operating system.

Pascal-3 V2.1D/RT-11 Release Notes

Changes in the Software

Version 2.1D is primarily a maintenance up-date; changes in software between Versions 2.1D and 2.1C consist for the most part of fixes for previously known problems.

Problems We've Fixed

Version 2.1D corrects the following problems:

Record field as a parameter

The compiler generated bad code when certain nested record fields were passed as a parameter.

Negative stack offset for common subexpressions

Depending upon the location of a variable's declaration, two identical calculations involving certain functions, including `sin` and `cos`, could have different results. The library routines for the functions caused the compiler to reference a negative offset from the stack pointer.

Wrong results from integer comparison

A comparison between an integer and a subrange that is an element of a packed record produced an incorrect result.

Incorrectly addressed real variable

Certain globally allocated real variables were incorrectly addressed. As a work-around, reducing the size of the program seems to remove the error.

Dynamic string package errors

The `Insert` routine printed the error message `Array subscript out of bounds` when the combined length of the two input strings exceeded the target string's length. Now, the insert string is concatenated onto the end of the target string and truncated.

The `Concatenate` routine reported errors when the string to be concatenated overflowed the target string. The input string is now truncated to fit.

%Page directive didn't work

On listings, `%page` incremented the page number incorrectly.

Reserved instruction trap error

Using a packed record containing integers in a nested procedure call caused a reserved instruction trap.

Illegal instruction gave "compiler writer error"

Using certain illegal instructions resulted in the error message for internal problems instead of the appropriate error message.

No error message

The compiler failed to give an error message for attempts to pass an element of a packed structure as a variable-parameter.

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

Problems We've Fixed

The compiler incorrectly used R0 for procedure calls

The compiler failed to save the contents of the R0 register after nonpascal procedure calls.

Memory protection violation

A memory protection violation occurred in isolated instances when the value of R0 was not initialized before adding the offset.

Sets generated bad code

Set constructor expressions of the form [var1..var2] generated bad code at times.

Set type produces wrong results

The declaration and statement

```
type  a = set of (one,two,three);  
var   b:a;
```

```
begin  
    b:= [one,three];  
    write(b * [] <= []);  
end.
```

produced a set of 256 items merged with a set of 3 results.

Function returns incorrect result

The compiler incorrectly changed bit lengths to byte lengths when a function returned a structured type.

Bit optimisations incorrect

Compiler optimization of certain user-defined procedure calls produced an incorrect sequence of execution.

PB formatter rejects "nonpascal"

The formatter failed to recognize the nonpascal directive. It now treats nonpascal exactly like the external directive.

String comparison range wrong

For string comparisons, ord(char) was in the range 0..255 instead of -128..127. The range is now correct.

Unsigned characters treated as signed

The compiler treated 8-bit characters as if they were signed numbers when the character was optimized.

Assignment statements caused failure

A complicated series of assignment statements involving arrays of type real previously failed.

Compiler consistency checks reported

Certain instances of undeleted temps in procedure... were fixed in 2.1D.

1890

The following is a list of the names of the persons who have been elected to the office of the Board of Directors of the City of New York for the year 1890.

The names of the persons who have been elected to the office of the Board of Directors of the City of New York for the year 1890 are as follows:

1. John A. B. Smith

2. John A. B. Smith

3. John A. B. Smith

4. John A. B. Smith

5. John A. B. Smith

6. John A. B. Smith

7. John A. B. Smith

8. John A. B. Smith

Pascal-3 V3.1D/RT-11 Release Notes

Linkage failure during installation

A missing continuation character in the file UTILS.XM, which builds the utilities, caused a failure at the point where XREF is linked.

Misleading error message for nonexistent files

The compiler printed the error message "Unknown Pascal run-time error" for an attempt to compile a nonexistent file or a file containing a `$include` of a nonexistent file.

Debugger overlays do not work for XM monitor systems

XMDBG.COM, the XM monitor command file that links Debugger modules with a Pascal program, now correctly overlays Debugger modules.

THE UNIVERSITY OF CHICAGO
THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO
THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO
THE UNIVERSITY OF CHICAGO

Documentation Notes

The documentation for Pascal-2 Version 2.1D includes the second edition of the *Pascal-2 User Manual*, which documents the enhanced V2.1 software, and Update Package No. 2, documenting the 2.1D features. (Update Package No. 1 was previously distributed and all changes then noted have been incorporated into the manual.) We have expanded a number of sections, including "External Modules," "Resident and Cluster Libraries," and "The Debugger Guide." The expanded sections provide new information and clarify and correct earlier material, at user suggestion. Update Package No. 2, records these changes and is made up of "change pages" for insertion into the user manual at the appropriate place. The update package's cover sheet should be inserted into the manual just before the first contents page, to keep a history of changes to the manual.

The *Release Notes*, containing the Installation Guide, is a companion document of the *Pascal-2 User Manual* and should be kept with that manual for reference.

Miscellaneous Notes

Official ISO Standard for Pascal

The Pascal language now officially has an international standard. The International Standards Organization, in a vote tallied this summer, adopted a standard language definition for Pascal. The action followed earlier adoption of an American standard that is a subset of the international one, lacking only conformant array parameters.

The sequence leading to adoption of the Pascal standard began in December 1982, when ANSI and IEEE agreed on the American standard, identical to the international draft standard except for conformant array parameters. At the same time, the Joint Pascal Committee of ANSI and IEEE recommended adoption of the international standard Level 1 (including conformant array parameters) to the U.S. committee known as X3J9, which then voted "yes" on the international standard at the next meeting. Previously, the U.S. was one of three "no" votes. This time, the ISO standard passed with no dissenting votes and one abstention.

Peppered throughout the Language Specification of the user manual are references to the "draft" standard. The word "draft" can be ignored. (Change pages were not issued for this change.)

Packing Problems

Several users have uncovered bugs in Pascal-2's packing of records and arrays. We believe that it may be necessary to modify the packing algorithm slightly in version 2.1E in order to correct these problems. If such a change is necessary, users who have programs that involve the reading and writing of packed records or arrays may have to convert their data files to the new format in order to use V2.1E. This new version will be released in roughly six months; if the packing conversion is necessary we will make every effort to clearly document when and how to follow the conversion process.

Stack Overflows

Especially large stack overflows, those that extend beyond the heap and into the program code sections of memory, have the potential for causing serious problems. In some cases it is possible for the error to prevent the appropriate error message from being printed or the program from properly terminating. In some rare instances, the condition can even lead to the disruption of the computer's operating system. It is the programmer's responsibility to avoid such excessively large overflows by controlling the size of local variables and value parameters passed to procedures.

The first part of the report deals with the general situation of the country. It is a very interesting and informative study of the country's development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's development.

The second part of the report deals with the economic situation of the country. It is a very interesting and informative study of the country's economic development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's economic development.

The third part of the report deals with the social situation of the country. It is a very interesting and informative study of the country's social development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's social development.

The fourth part of the report deals with the political situation of the country. It is a very interesting and informative study of the country's political development. The author has done a great deal of research and has gathered a wealth of material. The report is well written and is a valuable contribution to the study of the country's political development.

EXITST Creates a Walkback for 'Severe Error' Status (4)

The `Exitst` procedure is a support library routine that sets the termination status of a program and stops the program when a "severe error" status is detected. The procedure's integer argument determines the termination status for any program that calls it. When a "severe error" status of 4 is passed, the procedure also invokes the post mortem analyzer to create a walkback of the program execution from the point of failure.

Handwritten text, likely a section header or title.

Handwritten text block, possibly a paragraph or list of items.



Pascal-2 V2.1D/RT-11 Installation Guide

This guide describes the way to load the RT-11 Pascal-2 software on your system. The V2.1 software runs on RT-11 V4 and V5.

To install Pascal-2 V2.1 for RT-11, follow the steps below:

1. Copy all of the Pascal-2 files to the system device (SY:);
2. Select a compiler depending on your system monitor;
3. Select a run-time library depending on your processor hardware options;
4. Compile the Pascal-2 utility programs;
5. Set the compiler's listing file page size to a value other than 66 octal (optional);
6. Extract the Debugger modules from the support library for use in overlaying the Debugger against user programs.
7. Delete files no longer needed (optional). See Appendix B for a sample deletion command.

These steps are described in detail in the following paragraphs, and are illustrated by an example that can be found in Appendix B of this guide. The installer should read this guide in its entirety before attempting to install the software. Upon successful completion of the installation, the Pascal-2 system is fully operational as described in the *Pascal-2 User Manual*.

This guide contains two appendices. Appendix A lists the files contained in the release media. Appendix B shows a typical XM installation.

Copying the Pascal-2 Files to the System Device

Copy all of the distribution files to the system device (SY:) using the **ASSIGN** and **COPY** commands, as shown below:

```
.ASSIGN SY DK _____ assigns SY: as default device  
.COPY MTO: SY: _____ copies magtape to system device
```

If your system disk is a floppy disk, or is otherwise limited in available space, you should first read the following sections and select the files that are necessary for your system. Then copy those files to your system. The minimal system requires one compiler file and one library file. You may wish to build more than one system disk and, for example, install the compiler, library, and Debugger on one disk and the utility programs on a second system disk.

Selecting a Compiler for Your System Monitor

There are two compilers supplied with Pascal-2. Your choice of a compiler depends on the version of the RT-11 monitor you intend to use. There are four possibilities: the Base-Line (BL) monitor, the Single-Job (SJ) monitor, the Foreground-Background (FB) monitor, and the eXtended-Memory (XM) monitor.

If you are using either the BL or the SJ monitor, choose the compiler called SJ.SAV. If you use the XM monitor, select the compiler called XM.SAV.

The FB monitor does not leave sufficient memory to run the Pascal-2 compiler. If you are using the FB monitor, you must switch to either the SJ or XM monitor (using the **BOOT** command) before compiling a Pascal program. Once compiled, programs may be linked and run under the FB monitor.

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

NEW YORK: PUBLISHED BY J. B. LIPPINCOTT & CO.

1850

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

NEW YORK: PUBLISHED BY J. B. LIPPINCOTT & CO.

1850

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

NEW YORK: PUBLISHED BY J. B. LIPPINCOTT & CO.

1850

THE HISTORY OF THE UNITED STATES

OF THE UNITED STATES OF AMERICA

FROM THE FIRST SETTLEMENTS TO THE PRESENT TIME

BY JAMES M. SMITH

Pascal-2 V2.1D/RT-11 Installation Guide

The XM compiler should be chosen over the SJ compiler where possible because it uses the extended memory or "virtual" overlay capability and gives faster compilations.

When you have selected a compiler file, copy it to SY:PASCAL.SAV with the following command, which assumes the default device is SY:.

.COPY SJ.SAV PASCAL.SAV _____ installs SJ compiler
or:
.COPY XM.SAV PASCAL.SAV _____ installs XM compiler

You may then delete SJ.SAV and XM.SAV, or you may leave them on your system disk for use under their respective monitors. (See Appendix B.)

Selecting a Run-Time Library

There are four run-time libraries supplied with Pascal-2, one for each combination of processor instruction sets. Choose the library that matches the configuration of the processor that will run your compiled programs.

The possible configurations are:

- FPP — a processor with the Floating Point Processor instruction set. The FPP is standard equipment on the PDP-11/60 and optional on all new PDP-11's and the LSI-11/23. If your processor includes the FPP, select the LIBFPP.OBJ library.
- FIS — the Floating Instruction Set. The FIS hardware is an option available for the LSI-11, LSI-11/2 and some older PDP-11 processors. If your processor has FIS, select the LIBFIS.OBJ library.
- EIS — Extended Instruction Set, for hardware support of multiply, divide, and long shift instructions. EIS is standard equipment on all new PDP-11 and LSI-11/23 processors and an option available for all older LSI-11's and PDP-11's. If your processor has neither FPP nor FIS, but does have EIS, then select the LIBEIS.OBJ library.
- For processors with no extended or floating instructions, select the LIBSIM.OBJ library. This library operates on any LSI-11 or PDP-11 regardless of its actual configuration, but does not take advantage of any optional hardware.

After selecting a library file, copy it to SY:PASCAL.OBJ with the following command or one similar to it, depending on the processor configuration. In the command, SY: is the default device.

.COPY LIBFPP.OBJ PASCAL.OBJ _____ FPP library

You may then remove the other library files, or leave them on the system for use with other configurations. (See Appendix B.)

Compiling the Utility Programs

Five Pascal-2 utility programs are supplied in source form and are automatically compiled and linked by one of two command files, UTILS.SJ and UTILS.XM. UTILS.SJ prepares the utility programs for execution under the SJ, FB and BL monitors; UTILS.XM prepares them for execution under XM. Feel free to examine these command files before executing it indirectly using one of two commands:

.UTILS.SJ
or:
.UTILS.XM

...the ...
...the ...
...the ...
...the ...
...the ...

...the ...
...the ...

...the ...
...the ...
...the ...
...the ...

...the ...
...the ...
...the ...
...the ...
...the ...

...the ...
...the ...
...the ...
...the ...
...the ...

...the ...
...the ...
...the ...
...the ...
...the ...

...the ...
...the ...

Compiling the Utility Programs

Upon completion of UTILS, the utility programs are available for use as described in the Utilities Guide of the *Pascal-2 User Manual*.

In UTILS.XM all of the utilities must be linked as "virtual jobs" made up of virtual overlays because their load images are larger than 16K. (The XM monitor can not load a program with a root segment larger than 16K.) The Pascal-2 file START.OBJ is used in the link process to create a null root segment, tricking the Linker into loading the oversized program. The *Pascal-2 User Manual* contains a detailed description of this technique for user programs in Pascal.

When UTILS completes, you may then delete the extra .OBJ and .MAP files left on the disk by the UTILS command file. (See Appendix B.)

Setting the Page Size of the Compiler's Listing File

The command files RTPAGE.COM and XMPAGE.COM, supplied with the Pascal-2 release, provide a means for changing the number of lines per page of the SJ and XM compilers' listing file, respectively. This feature allows for the use of odd-sized printer paper or for special printing needs.

Before executing either command file, you must modify the lines-per-page value in the fifth line of data in the command file. (The first line of data is the file to patch.) The normal page size (in octal) is 66, not counting header lines.

```
! Command file to patch the SJ version of Pascal to change
! the lines per page of a listing file
!
! Change the number nn (octal) to another value, in octal
! that represents the number of lines per page of a listing file
! that you require.
!
R SIPP
SY:PASCAL

3075
2
nn _____ lines per page
^Y
^C
```

where *nn* is the desired page size in octal. The value '3075' is the base address of the symbol PAGELN, and the actual value in the supplied command file may be different. The value '2' is the byte offset from the base address of the location to patch. Values for the page size and byte offset must not be changed.

After editing the appropriate file, execute it indirectly with the command:

```
RTPAGE _____ for SJ installation
or:
XMPAGE _____ for XM installation
```

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

Pascal-2 V2.1D/RT-11 Installation Guide

Extracting Support Library Modules for Debugger Overlays

Before users can overlay the Debugger against their programs, the command file EXTRAC.COM must be executed. This command file, supplied with the release kit, extracts from the Pascal support library the Debugger and other key modules used by the XMDBG.COM (for XM systems) and SJDBG.COM (for SJ systems) command files, also in the release kit. (Users can then execute either XMDBG.COM or SJDBG.COM to overlay the Debugger and their Pascal programs.) EXTRAC.COM places the modules onto the system device (SY:), with the extension .DBG. It need only be executed once.

The command file can be executed with the indirect (at-sign) processor, as shown:

.@EXTRAC

Installing Pascal-2 With Pascal-1

The Pascal-1 system compiler can be renamed to allow simultaneous use of Pascal-1 and Pascal-2. Then, the files for both systems can be present on the system device without conflict, provided that the compiler/library versions and upgrade levels are themselves compatible. (For example, Pascal-1 V1.2K is compatible with Pascal-2 V2.0K, and V1.3C is compatible with V2.1C; V1.2K, however, is not compatible with V1.3C.

The object library PASCAL.OBJ can be shared by the two systems (as long as the two system versions are compatible); the libraries supplied with Pascal-2 include all of the Pascal-1 routines.

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS

MEMORANDUM FOR THE RECORD
SUBJECT: [Illegible]
DATE: [Illegible]
BY: [Illegible]

1. [Illegible]

2. [Illegible]

3. [Illegible]

4. [Illegible]

Appendix A: Distribution Files

Compilers

SJ .SAV Pascal-2 Compiler for SJ monitor
 XM .SAV Pascal-2 Compiler for XM monitor

Object Libraries

LIBFPP.OBJ Library for processors with FPP and EIS
 LIBFIS.OBJ Library for processors with FIS and EIS
 LIBEIS.OBJ Library for processors with EIS only
 LIBSIM.OBJ Library for base-level processors
 VIRJOB.OBJ Header module for XM virtual jobs
 START .OBJ Null root module for XM virtual overlays

Command Files

EXTRAC.COM Support library module extractor for Debugger overlays
 SJDBG .COM Debugger overlayer for SJ monitor
 XMDBG .COM Debugger overlayer for XM monitor
 RTPAGE.COM SJ listing-file page size patcher
 XMPAGE.COM XM listing-file page size patcher
 UTILS .SJ SJ, FB and BL utility installer
 UTILS .XM XM utility installer

Utility Programs

PASMAT.PAS Program formatter
 PB .PAS Pascal Beautifier
 XREF .PAS Cross-referencer
 PROCREF.PAS Procedure cross-referencer
 PROSE .PAS Text formatter
 STRING.PAS String package
 PASMAL.MAC MACRO-11 interface package
 SAYERR.PAS System error message printer
 UERRON.PAS User error reporting module
 OPERRO.PAS Support library error routine
 LIBDEF.PAS Support library definitions
 CSITYP.PAS Command String Interpreter (CSI) type definitions
 CSIPRO.PAS CSI procedure definitions
 FIXARG.PAS CSI argument parser
 FIXINC.PAS CSI %include file handler
 FIXOUT.PAS CSI temporary output manager
 SYMDCL.PAS CSI symbol table declarations
 SYMCOO.PAS CSI symbol table manager
 CIVNUM.PAS CSI command-line number converter

Demonstration Programs

* Vary from release to release

1940-1941

1940-1941

1940-1941

1940-1941

1940-1941

Pascal-2 V2.1D/RT-11 Installation Guide

Appendix B: Typical XM Installation

The following steps illustrate the installation of Pascal-2 from magtape on a processor that includes the FPP floating point processor. The Pascal-2 compiler for the extended memory (XM) monitor is selected.

```
.ASSIGN SY DK _____ assigns SY: as the default device
.COPY MTO: SY:
.COPY LIBFPP.OBJ PASCAL.OBJ _____ selects the FPP library
.COPY XM.SAV PASCAL.SAV _____ selects the XM compiler
.UTILS.XM _____ builds the utilities
R PASCAL _____ builds PASMAT
PASMAT/TIME/WORKSPACE=650
! The following link will round up the root to 4K and put the extra
! memory on the heap
R LINK
PASMAT,PASMAT=START,VIRJOB/U:20000//
PASMAT,PASCAL/V:1
//
P$GROW
~C
R PASCAL _____ builds PB
PB/TIME
R LINK
PB,PB=STRAT,VIRJOB/U:20000//
PB,PASCAL
//P$GROW
~C
R PASCAL _____ builds XREF
XREF/TIME
R LINK
XREF=START,VIRJOB/U:20000//
XREF,PASCAL/V:1
//
P$GROW
~C
R PASCAL _____ builds PROCREF
PROCREF/TIME
R LINK
PROCREF,PROCREF=START,VIRJOB/U:20000//
PROCREF,PASCAL/V:1
//
P$GROW
~C
R PASCAL _____ builds PROSE
PROSE/TIME/WORKSPACE=700
R LINK
PROSE,PROSE=START,VIRJOB/U:20000//
PROSE,PASCAL/V:1
//
P$GROW
~C
```

1890

Received of the Treasurer of the County of ... the sum of ...

for ...

...

...

...

...

...

...

...

Appendix B: Typical XM Installation

.QMPAGE _____ sets listing page size to value other than 66 octal (optional)
.QEXTRAC _____ extracts library modules for Debugger overlaying
.R PIP _____ cleans up the system disk (optional)
*SJ.SAV.XM.SAV/D _____ Remove the compilers, leaving PASCAL.SAV
*LIBFPP.OBJ.LIBFIS.OBJ.LIBEIS.OBJ.LIBSIM.OBJ/D
*PASMAT.OBJ.PB.OBJ.XREF.OBJ.PROCRE.OBJ.PROSE.OBJ/D
*PASMAT.MAP.PB.MAP.XREF.MAP.PROCRE.MAP.PROSE.MAP/D
*_C

Page 100

The first part of the paper is devoted to a discussion of the general principles of the theory of the structure of the atom. It is shown that the structure of the atom is determined by the laws of quantum mechanics, and that the laws of quantum mechanics are in agreement with the experimental facts.

DISTRIBUTOR CUSTOMERS

Please submit all technical support questions, verbal or written, directly to your authorized Oregon Software Distributor.

When you have a problem, please contact your Distributor by telephone. If the problem cannot be solved over the telephone, we request that you submit a written Trouble Report.

When submitting a written report, please complete one of the enclosed Trouble Report Forms and send it to your Distributor along with a small sample program demonstrating the problem you are experiencing. If you cannot reduce the problem to a one-page sample program. Submit your Trouble Report with the program on a single density floppy disk (preferably RT-11 format) or on a magtape. Your Distributor will then duplicate the problem at his facility.

In most cases your Distributor will be able to answer your questions. If necessary, your Distributor will forward your written report to Oregon Software for a solution. We will do everything possible to respond to your problem within 30 days after the report is received at Oregon Software. Once a solution is reached, it will be forwarded to your Distributor who will then contact you.

Thank you for support and interest in Oregon Software products.

The first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

the first of these is the fact that the
the first of these is the fact that the
the first of these is the fact that the

INSTRUCTIONS

Check recent issues of our Pascal Newsletter to see whether the trouble has been reported. The Newsletter may describe a quick fix or tell you that the problem has been fixed in the current release. If so, your Designated Contact Person should request an update in writing. If our Newsletter and manuals don't help, complete the front of this Trouble Report and send it to Oregon Software. A few suggestions:

Your Name: We'll call or write you if we need more details about the problem.

Designated Contact Person: The "responsible official" for our software at your site, usually the senior programmer. We'll send the DCP a copy of the correspondence about this problem.

License Number: The serial number of your site printed in the headline of all program listings. Looks like "1-876."

Some problems relate to specific combinations of hardware, software, and processors. We need to know your setup:

Processor Model: The processor line and specific model number (*not* the serial number!).

Examples: *PDP-11/44, 256 KB • LSI-11/23, 56 KB • VAX-11/750, 768 KB.*

Processor Options: Any installed processor options that extend the instruction set. EIS = Extended Instruction Set. FIS = Floating Instruction Set. FPP = Floating-Point Processor. FPA = Floating-Point Accelerator.

Operating System and Version: The software operating environment. Please, *please*, include the specific version! Examples: *RT-11 V4 XM • RSX-11M-Plus V3.2 • RSTS/E V7A • VMS V2.1*

Problem Description: Reduce the problem to the simplest situation in which it occurs. One page is the largest program we can reasonably type by hand. If a larger amount of code is needed to demonstrate the error, send us a magtape or single density floppy. Unfortunately, we cannot return these. Note any intuitions you have about possible causes of the trouble. Remember: WE MUST BE ABLE TO REPRODUCE THE TROUBLE HERE.

OUR RESPONSE

It's already fixed or Fixed in next release.

You will automatically receive a copy of a release in which the problem is fixed, if your Designated Contact Person has signed the Conditional Update Request on the front of this form.

Maybe a bug, but ...

Documentation problems, customer errors, obscure "features" or inherent limitations in our implementation can each cause trouble. We will help you work around the problem and/or supplement our documentation in these cases. Please understand that our support is limited to correcting problems with our software and does not include applications programming.

In any case—

We will send you a written response within a month.

Trouble Report

See instructions on reverse side.

CUSTOMER IDENTIFICATION

Your Name _____ Designated Contact Person (if different) _____

Conditional Update Request: If the Trouble Report pertains to a problem that has been fixed or will be fixed in the next release, this signature constitutes a request for an update which Oregon Software will provide under the terms of the existing software license and/or support agreements.

Signature _____ Date _____ / _____ / _____

Company _____ License Number _____

Address _____ Date Submitted _____ / _____ / _____

Telephone () _____ ext. _____

City _____ State _____ Country _____ Zip/Postal Code _____

Processor Model _____ Processor Options (EIS, FIS, FPP, FPA, etc.) _____

Distribution Medium _____ Attachments (Listings, Magtape, etc.) _____

Software and Version _____ Operating System and Version _____

PROBLEM DESCRIPTION

☐ Major Problem ☐ Minor Problem ☐ Suggestion ☐ Inquiry

If the problem could have been avoided by better documentation, explain below.

If the problem is intermittent, outline below the conditions under which the problem seems to occur.

Describe the problem (use additional sheets if necessary):

FOR OREGON SOFTWARE USE

Log Number _____ Title _____

Date Assigned _____ / _____ / _____ Problem Type _____

Assigned to _____

Date of Response to Customer _____ / _____ / _____ Summary of Action _____

☐ Circulate _____

Oregon Software

6915 S.W. Macadam Avenue
Portland, Oregon 97201, USA
503/245-2202
TWX: 910-464-4779

INSTRUCTIONS

Check recent issues of our Pascal Newsletter to see whether the trouble has been reported. The Newsletter may describe a quick fix or tell you that the problem has been fixed in the current release. If so, your Designated Contact Person should request an update in writing. If our Newsletter and manuals don't help, complete the front of this Trouble Report and send it to Oregon Software. A few suggestions:

Your Name: We'll call or write you if we need more details about the problem.

Designated Contact Person: The "responsible official" for our software at your site, usually the senior programmer. We'll send the DCP a copy of the correspondence about this problem.

License Number: The serial number of your site printed in the headline of all program listings. Looks like "1-876."

Some problems relate to specific combinations of hardware, software, and processors. We need to know your setup:

Processor Model: The processor line and specific model number (*not* the serial number!).

Examples: PDP-11/44, 256 KB • LSI-11/23, 56 KB • VAX-11/750, 768 KB.

Processor Options: Any installed processor options that extend the instruction set. EIS = Extended Instruction Set. FIS = Floating Instruction Set. FPP = Floating-Point Processor. FPA = Floating-Point Accelerator.

Operating System and Version: The software operating environment. Please, *please*, include the specific version! Examples: RT-11 V4 XM • RSX-11M-Plus V3.2 • RSTS/E V7A • VMS V2.1

Problem Description: Reduce the problem to the simplest situation in which it occurs. One page is the largest program we can reasonably type by hand. If a larger amount of code is needed to demonstrate the error, send us a magtape or single density floppy. Unfortunately, we cannot return these. Note any intuitions you have about possible causes of the trouble. Remember: WE MUST BE ABLE TO REPRODUCE THE TROUBLE HERE.

OUR RESPONSE

It's already fixed or Fixed in next release.

You will automatically receive a copy of a release in which the problem is fixed, if your Designated Contact Person has signed the Conditional Update Request on the front of this form.

Maybe a bug, but ...

Documentation problems, customer errors, obscure "features" or inherent limitations in our implementation can each cause trouble. We will help you work around the problem and/or supplement our documentation in these cases. Please understand that our support is limited to correcting problems with our software and does not include applications programming.

In any case—

We will send you a written response within a month.

Trouble Report

See instructions on reverse side.

CUSTOMER IDENTIFICATION

Your Name _____ Designated Contact Person (if different) _____

Conditional Update Request: If the Trouble Report pertains to a problem that has been fixed or will be fixed in the next release, this signature constitutes a request for an update which Oregon Software will provide under the terms of the existing software license and/or support agreements.

Signature _____ Date _____/_____/_____

Company _____ License Number _____

Address _____ Date Submitted _____/_____/_____

Telephone () _____ ext. _____

City _____ State _____ Country _____ Zip/Postal Code _____

Processor Model _____ Processor Options (EIS, FIS, FPP, FPA, etc.) _____

Distribution Medium _____ Attachments (Listings, Magtape, etc.) _____

Software and Version _____ Operating System and Version _____

PROBLEM DESCRIPTION

☐ Major Problem ☐ Minor Problem ☐ Suggestion ☐ Inquiry

If the problem could have been avoided by better documentation, explain below.

If the problem is intermittent, outline below the conditions under which the problem seems to occur.

Describe the problem (use additional sheets if necessary):

FOR OREGON SOFTWARE USE

Log Number _____ Title _____

Date Assigned _____/_____/_____ Problem Type _____

Assigned to _____

Date of Response to Customer _____/_____/_____ Summary of Action _____

☐ Circulate _____

Oregon Software

6915 S.W. Macadam Avenue
Portland, Oregon 97201, USA
503/245-2202
TWX: 910-464-4779

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO
DIVISION OF THE PHYSICAL SCIENCES
DEPARTMENT OF CHEMISTRY
5408 S. DICKINSON AVE.
CHICAGO, ILL. 60637
TEL. 733-4331
FAX 733-4331
WWW.CHEM.UCHICAGO.EDU

CHICAGO, ILL. 60637
TEL. 733-4331
FAX 733-4331
WWW.CHEM.UCHICAGO.EDU